A Self-Healing Framework for General Software Systems

Doctoral Dissertation submitted to the
Faculty of Informatics of the *Università della Svizzera Italiana*
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy

presented by
Nicolò Perino

under the supervision of
Prof. Mauro Pezzè

August 2014

Dissertation Committee

Prof. Matthias Hauswirth	Università della Svizzera Italiana, Switzerland
Prof. Nate Nystrom	Università della Svizzera Italiana, Switzerland
Prof. Oscar Nierstrasz	University of Bern, Switzerland
Prof. Sebastian Uchitel	Imperial College London, United Kingdom Universidad de Buenos Aires, Argentina

Dissertation accepted on 31 August 2014

Prof. Mauro Pezzè
Research Advisor
Università della Svizzera Italiana, Switzerland

Prof. Stefan Wolf
PhD Program Director

Prof. Igor Pivkin
PhD Program Director

I certify that except where due acknowledgement has been given, the work presented in this thesis is that of the author alone; the work has not been submitted previously, in whole or in part, to qualify for any other academic award; and the content of the thesis is the result of work which has been carried out since the official commencement date of the approved research program.

Nicolò Perino
Lugano, 31 August 2014

Abstract

Modern systems must guarantee high reliability, availability, and efficiency. Their complexity, exacerbated by the dynamic integration with other systems, the use of third-party services and the various different environments where they run, challenges development practices, tools and testing techniques. Testing cannot identify and remove all possible faults, thus faulty conditions may escape verification and validation activities and manifest themselves only after the system deployment. To cope with those failures, researchers have proposed the concept of *self-healing* systems. Such systems have the ability to examine their failures and to automatically take corrective actions. The idea is to create software systems that can integrate the knowledge that is needed to compensate for the effects of their imperfections. This knowledge is usually codified into the systems in the form of redundancy. Redundancy can be deliberately added into the systems as part of the design and the development process, as it occurs for many fault tolerance techniques. Although this kind of redundancy is widely applied, especially for safety-critical systems, it is however generally expensive to be used for common use software systems.

We have some evidence that modern software systems are characterized by a different type of redundancy, which is not deliberately introduced but is naturally present due to the modern modular software design. We call it *intrinsic redundancy*. This thesis proposes a way to use the intrinsic redundancy of software systems to increase their reliability at a low cost. We first study the nature of the intrinsic redundancy to demonstrate that it actually exists. We then propose a way to express and encode such redundancy and an approach, *Java Automatic Workaround*, to exploit it automatically and at runtime to avoid system failures. Fundamentally, the Java Automatic Workaround approach replaces some failing operations with other alternative operations that are semantically equivalent in terms of the expected results and in the developer's intent, but that they might have some syntactic difference that can ultimately overcome the failure. We qualitatively discuss the reasons of the presence of the intrinsic redundancy and we quantitatively study four large libraries to show that such redundancy is indeed a characteristic of modern software systems. We then develop the approach into a prototype and we evaluate it with four open source applications. Our studies show that the approach effectively exploits the intrinsic redundancy in avoiding failures automatically and at runtime.

Contents

Figures

Tables

Listings

Grammars

Chapter 1

Introduction

Reliability has always been one of the major challenges for software systems, historically for safety-critical systems, and today it is amplified by the pervasiveness of software systems in everyday activities.

High reliability is important, indeed many techniques have been proposed to increase it, but very difficult to achieve, indeed software systems still fail at runtime. The constant expanding complexity of software systems, which are intertwined with other systems and dependent on third-party systems, challenges all the usual development practices, tools and testing techniques, which cannot assure to release systems without faults. The increasing dimensions and complexity of modern systems, and thus, the growing testing costs make unrealistic the idea to avoid all failures before the deployment.

The classic maintenance cycle, which requires a failure report, an offline fixing by the developers, and a new release and deployment of the system, is still required but it is becoming insufficient. Maintaining systems offline and fixing faults are still necessary activities, but they result in a discontinuity in the services provided by the systems. In between the time a failure is reported and the time the fix is released by the developers, the functionalities of the system are not fully available. In this *gray* time slot, the system may not be usable at all or it may be only partially usable. In both cases, failures lead the system to unexpected behaviors, which may span from wrong results to incomplete actions or system crashes. In such cases, the users can avoid to use the system, or can tolerate an instable software, eluding failures with manual workarounds. As an extreme example, consider the fault 3655 of the Firefox browser, which was reported for the first time in March 1999, and still, after numerous fix attempts, it is still open after fourteen years[1].

To complement the classic maintenance cycle, researchers have proposed several mechanisms to tolerate or mask failures. Since the seventies, typically for safety-critical systems, this has been done through fault tolerance systems that increase the reliability

[1] `https://bugzilla.mozilla.org/show_bug.cgi?id=3655`

adding redundancy into the system, either hardware or software. As software redundancy, the most successful fault tolerance techniques are N-Version Programming and Recovery Blocks, developed by Avizienies [Avi85] and Randall [Ran75], respectively. These techniques are based on the idea that developing multiple versions of the same component reduces the probability of introducing the same faults in every version, and it also reduces the probability to get the same failure for the same input on the different versions. The different versions run in parallel, for N-Version Programming, or in sequence, for Recovery Blocks. The outcome is determined by the output of the majority of the executions, for N-version, and by the first non-failing execution, for Recovery Blocks.

In such techniques redundancy is deliberately added at design time into the system. This approach leads to more reliable software, but it incurs higher design, development, test, and maintenance costs, as well in a stricter and more rigorous process. For safety critical systems, higher costs are justified by the high standards of reliability that the systems must guarantee, but not for common use software.

For this reason, recently, autonomic computing and self-managed systems have emerged as a possible way of coping efficiently with runtime problems, at a lower cost. The idea has been introduced in the beginning of the year 2000 by Horn and Kephart [Hor01, KC03]. The underlying idea of autonomic computing is that systems are aware of themselves. They can reconfigure themselves, recover from failures, optimize for better performance, and protect themselves from external attacks. To achieve these goals, self-managed systems are built on a cycle that operates as a controller to monitor the system, analyze its behavior, plan possible changes to adapt, and execute the selected plan.

As part of self-managed systems, self-healing systems usually deal with failures that deteriorate system functionality or performance. Several techniques have been studied and proposed by many researchers. For example Demsky *et al.* propose an approach based on *recovery tasks*. A recovery task is a function that can be executed even if a failure causes some of its parameters to be unavailable. When a failure occurs, the technique uses an ad-hoc written recovery algorithm and static analysis to understand which recovery task must be taken to recover from the failure and continue the execution [DZM10]. Samimi *et al.*, instead, refer to formal specification as a reliable alternative to the implementation. On a failure, they use a constraint solver to execute the specification to allow the program to continue properly [SAM10]. Several self-healing approaches have been proposed for the Web. Gurguis *et al.* propose to apply the autonomic cycle proposed by Kephart *et al.* to determine problems in the execution of a service and use Web service discovering mechanisms to select a different service that can overtake the problem [GZ05]. Similarly, Guinea proposes an approach based on recovery strategies to allow a system to continue its execution when a misbehavior have been discovered. One of the proposed strategies substitutes the faulty service with a redundant one [Gui05].

Most of the techniques proposed so far are less expensive in terms of design, development, and process costs, and are usually preferred when the criticality of the system does not concern the safety. Still, these techniques make strong assumptions, require additional costs, or are limited in the scope. For example some techniques require specification or to write extra code, which means additional design and development costs. Other techniques are limited in the domain, for example the Web domain.

In this thesis we propose a new self-healing mechanism, called *Java Automatic Workaround* (JAW). This mechanism does not have domain limitation, it is widely applicable to different software systems, and does not incur high additional costs. The underlying idea is based on the intuition that the complexity of modern software systems and of their design and develop processes make them already intrinsically redundant. With the term *intrinsic redundancy* we define that kind of software redundancy that arise due to the modern methods of designing, engineering, and developing software systems rather then the specific purpose of replicating the functionality of a system. Thus, we can exploit the redundancy that is intrinsic in software systems to improve systems dependability and reliability, by capturing it in a suitable form and using it to avoid failures, instead of intentionally adding it during or after the development phase, or looking for it outside of the systems themselves. Redundancy can be found on different levels and can be either disadvantageous or acceptable. The former may derive from a bad design or maintenance. In the area of code cloning identification, several works have been done to identify and remove this kind of redundancy [KSNM05, KKI02, JMSG07]. We, on the other hand, exploit the good kind of redundancy. The reasons of the presence of the intrinsic redundancy are multiple. The modern development process leads to the use of third-party software components: systems may include several third-party components, and it is likely that some of them offer the same or similar functionalities. Also, despite the best design and development intentions, developers often implement multiple functions with the same or a similar logic. On the other side, modern design approaches also yield to highly configurable and versatile modular systems, which increase the possibility of overlapping functionalities among modules. Maintenance is also a source of redundancy. For example the process of functions deprecation, originated from a re-design of the system or performance issues, leads to functions duplication. Containers or graphical libraries are typical examples of intrinsically redundant components. For example, adding a set of items one by one or with a single operation, re-drawing an interface or clearing and drawing it from scratch, restarting a service or stopping and starting it again, sorting a set with two different kind of algorithms, are all examples of operations that lead to the same results. Our approach exploits the intrinsic redundancy at method calls level, and we call such behaviorally equivalent methods, *equivalence*. Let us take two method in a bug-free context, we say that the two methods are *semantically equivalent* if their two outcome are indistinguishable.

In a previous work, we focused on Web applications [CGPP10a]. We demonstrated

that, in the Web domain, intrinsic redundancy, indeed, exists and can be exploited to automatically avoid failures. We noticed that often developers, facing with failures in Web libraries, propose to avoid such failures by applying workarounds to the faulty code, that is using a different way to achieve the same result. We proposed an approach, called *Automatic Workaround for Web Applications* (AWA), to automatically avoid failures by means of equivalent sequences. The domain we considered was composed of stateless Web applications. The tool we developed involved the user as an active actor of the healing process, thus, implying a stop in the service due to the actions taken by the user to pinpoint the failure and validate the proposed workarounds.

In this thesis we focus on the new problem of exploiting intrinsic redundancy to create self-healing systems in the broad domain of general or common use applications, where we cannot make assumptions of a stateless behavior, the user cannot be involved actively in the healing process, and we cannot stop the execution of the systems, that is a runtime healing is required. Considering a larger and more varying set of software systems, new challenges must be faced.

We have to investigate if intrinsic redundancy is a typical characteristic of Web applications only, or it can be found also in other domains. Since our goal is to exploit intrinsic redundancy, we want to propose a way to identify, collect, and express it for self-healing purposes. Once the intrinsic redundancy has been identified and collected, we want to investigate if it can be generalized to make common use software systems self-healing. To deal with system failures, we must be able to detect a failure, to assure consistency in case of failure, and adapt the system behavior automatically, efficiently and at runtime.

In this work we demonstrate that common use software systems are indeed intrinsically redundant. We show and discuss the results of a study conducted on several large and widely used software libraries, where we identified and collect the intrinsic redundancy of those libraries, and expressed it through our representation, called *rewriting rules*. We also prove that the intrinsic redundancy of software systems can be used to avoid failures, and we propose an approach to handle failures automatically and at runtime, solving the problem of the system state corruption. JAW is based on the Automatic Workaround technique [CGP08b]and it works as follow: When the software is running normally, the state of the system is regularly saved to assure state consistency in case of state corruption due to a failure. We assume that a failure detection mechanism is provided, for example assertions, and when a failure is caught, the last saved state is restored to avoid any side-effect, thus the execution is rolled back to that point. An equivalence is selected and applied to the code. The equivalence is selected among those that have a match in the code between the restored point and the failed instruction. The execution is then resumed, if the equivalence works as a workaround, the execution continues, otherwise another equivalence is applied. When all the equivalences that can be applied to that portion of code have been tried, the state of the system is rolled backed recursively, following the list of all saved states

backward. When all the restorable states have been restored and no more equivalences can be applied, the system exits with an error message.

1.1 Research Hypothesis and Contributions

The hypothesis of this dissertation is that *software systems are intrinsically redundant, that is they inherently provide multiple ways to perform the same operations. This redundancy can be captured, expressed and used for avoiding failures automatically and at runtime in common use software systems*, thus to achieve a better reliability in stateful software systems. In these systems a failure may corrupt the state, so we must assure system state consistency and, when the state is compromised, we must recover it consistently.

The main contribution of this thesis is to provide an approach to exploit the intrinsic redundancy of software systems to add self-healing capabilities to common use applications. To demonstrate the validity of the approach, a prototype of a framework to automatically avoid failures at runtime is developed and validated through a set of case studies.

We already demonstrated that a sub-domain of software systems, that is Web applications, are intrinsically redundant, therefore, we expect it is reasonable to broaden the first part of the hypothesis to more general software systems domain. So, by providing an approach and a tool based on it, we expect to understand if *common use software systems are intrinsic redundant*, *how much they are intrinsically redundant* and *where we can found this redundancy*. The redundancy must be *identified*, *extracted* and *represented* in a suitable form before being used. At this point we can *exploit it to avoid system failures*. Furthermore, in stateful systems, a failure may corrupt the state, thus we want to *recover a corrupted state* and avoid failures *automatically* and at *runtime*.

This work proposes an approach, JAW, that works potentially for every software system. It keeps the state of the system consistent by recovering it when a failure corrupts it, and uses the intrinsic redundancy, expressed as rewriting rules, to provide an automatic workaround to avoid the failure. JAW has also been implemented into a prototype for Java applications that can address failures automatically, efficiently, and at runtime. The prototype does not consider concurrency problems. A set of experimental results that demonstrate the efficacy of the approach is then proposed and discussed.

1.2 Structure of the Dissertation

The remainder of this dissertation is structured as follows:

- Chapter 2 provides an overview of techniques to increase system reliability. We examine several fault tolerance and self-healing techniques that rely on some

form of software redundancy.

- Chapter 3 describes our approach to automatically exploit intrinsic software redundancy. Starting with the use of intrinsic redundancy as workaround on a real example, we explain how to automate the process.
- Chapter 4 focuses on intrinsic software redundancy. We give qualitative reasoning on the existence of intrinsic redundancy in reusable components and we present some quantitative data as a result of an inspection of some large Java libraries. We formalize the syntax of the rewriting rules and we define their semantics.
- Chapter 5 presents a comparison between two major techniques for state handling, software transactional memory and checkpoint and recovery.
- Chapter 6 focuses on how JAW applies code transformation at runtime, and how it benefits from the state handling mechanisms to avoid faults located far from the failure.
- Chapter 7 describes the prototypal implementation used to evaluate the approach.
- Chapter 8 presents the evaluation study on some open source applications.
- Chapter 9 summarizes the contribution of this work, delineates the conclusions and the future directions.

Chapter 2

Guaranteeing Reliability at Runtime

Reliability is the ability of a system to consistently perform its intended or required function, when requested and without degradation or failures. Barlow and Proschan define reliability as the probability at specified time t, the system S is operating and will continue to operate for an interval of duration x [BP65]. Reliability is a crucial element in every system, for safety reasons in safety-critical systems, for business and social consequences in business-critical systems, or even for the users' expectation in common use systems. Full reliability could be ideally achieved by a fault-free system, but the current technology is far from being able to produce fault-free systems and may never be able to.

Achieving high reliability has always been challenging for all the software engineering disciplines. Researchers coped with this problem in several ways. *Prevention* is a first mechanism: design diversity, formal specifications, program validation, severe design processes and methodologies, tools to enforce programming principles and software reuse, may help engineers reduce the faults in the system during the design and development [Yu98, FSS02]. All the most efficient prevention techniques, however, cannot guarantee the absence of faults.

Fault removal techniques address faults that remain in the system despite the use of the best design and development practices. A lot of approaches, techniques and tools have been proposed in the past decades, from manual inspection of the code, to program verification techniques, such as automatic testing and analysis [YP08]. To achieve high quality standards, the costs related to the testing phases inflate, and even the most sophisticated testing approaches and tools cannot detect all the faults. Some components or part of the systems are impossible to verify, and some faults are difficult to reveal or may manifest themselves only under rare circumstances.

To cope with those faults that still remain in the code, researchers have proposed *failure avoidance* techniques. Starting from the seventies, a lot of fault tolerance techniques have been proposed to cope with failures, especially in safety-critical systems. Fault tolerance techniques usually require some form of redundancy. Redundancy can

be added in the code, in the data or in the environment. Developers might be required to develop the system, or parts of it, in multiple copies, duplicate system data, or replicate the environment. Introducing redundancy deliberately increases the reliability at high cost: code redundancy increases design, develop and maintenance costs; data and environment redundancy increase the cost of the infrastructures. Nonetheless, the responsiveness of the system might also be affected by redundancy. High costs are acceptable for systems where safety is a primary concern.

There is a large class of software systems, such as business-critical or user-critical systems, that have less critical requirements, but still relevant reliability requirements. For these systems, high development or running costs cannot be sustained. For this reason, in the last twenty years, after Kepahrt [KC03] and the IBM manifesto [IBM06], autonomic and self-healing approaches have been proposed to increase system reliability by dealing with failures automatically and at runtime.

In the following sections we provide a small survey on the main classic fault tolerance and self-healing techniques.

2.1 Software Fault Tolerance

The primarily goal of fault tolerance techniques is to produce reliable systems that are able to accomplish their tasks also under unpredicted conditions, such as a failure. Software fault tolerance is the ability of a system to detect a failure occurrence, recover from the effects and let the system provide the requested service.

To deal with software faults, software fault tolerance techniques took inspiration from the experience of hardware fault tolerance technique, and achieved reliability by deliberately introducing some redundancy in the system code. To reveal and tolerate software faults, several techniques have been proposed since the seventies. N-Version Programming, Recovery Blocks, and Pseudo-oracles, for example, rely on multiple versions of the same code or component [Avi85, Ran75, Wey82]. Similarly, exception handling and Self-Checking Programming use redundancy to react to erroneous behavior of the system with predefined actions [Goo75, YC75]. While some techniques make the code redundant, other focus on data redundancy, such as robust data structures and data diversity approaches [CPW72, TMB80].

The next paragraphs describe the mentioned classic fault tolerance techniques. Smith [Smi88] and Saha [Sah06] provide more detailed surveys on fault tolerance.

N-Version Programming The technique has been borrowed by the traditional hardware fault tolerance concept of N-way redundant hardware, and has been originally proposed by Avizienis *et al.* [Avi85]. In the N-Version Programming approach, the modules of the system are designed and developed with N independent versions and are executed in parallel. Each version performs the same task in a different way and then submits the outcome to a voter that decides the correct answer and returns it as

the result of the task. To overcome faults, the N versions must rely on the concept of design diversity: the different versions must be designed and develop independently, using different design and implementation techniques, different tool sets, different programming languages, and possibly different environments. A simple voting algorithm consists of comparing the outcomes and decides for the majority. In general, to guarantee a majority quorum, a module must be implemented in $2k + 1$ different versions, to tolerate up to k faults. N-version programming relies on multiple design and implementation processes, and is thus an expensive technique, but does not require explicit oracles, since the approach relies on a consensus mechanism to accept or reject an execution.

Recovery Blocks The Recovery Blocks method was proposed originally in the middle of seventies by Randell, and Horning *et al.* [Ran75, HLMSR74]. The mechanism is based on a language construct for encapsulating components or parts of programs that must execute reliably. As for N-version programming, each block is developed in several independent versions. The blocks are executed sequentially: when a block fails, a redundant one is selected and executed alternatively to the failed one. This process repeats until a block does not fail or no more alternative blocks are available. The Recovery Blocks mechanism uses acceptance tests as an oracle to detect failures at runtime, and relies on a rollback mechanism to recover the system to a consistent state before trying to execute a different block. It is very important that each block executes with the same state of the system as before the previous block has been executed. The main differences between Recovery Blocks and N-Version Programming are the sequential way to process the blocks and the explicit oracles to detect failures and trigger the recovery actions.

Consensus Recovery Block In the eighties, Scott proposed to combine N-Version Programming and Recovery Blocks to overcome the main limitations of the two approaches and proposed the Consensus Recovery Block approach [Sco83]. The major limitations of Recovery Blocks are the development of acceptance tests without a precise guideline, the lack of a testing methodology and the high cost of acceptance computation. N-Version Programming suffers from the difficulty of producing the voting system: it may happen that different versions compute different albeit correct results, or results with different degrees of precision. Scott proposed to implement n versions of the program or the component and use a mix of voting mechanism and acceptance tests for checking the results. The versions of the systems run in parallel and the results are submitted to the voting procedure. If two of the n results agree, their output is considered correct, otherwise each result is sequentially examined for acceptance and the first successful result is considered the correct one.

Pseudo-oracles The underlying idea behind N-Version Programming and Recovery Blocks has been exploited also in software testing. In 1982, Weyuker proposed the Pseudo-oracles approach to address the problem of *non-testable* programs, that is programs for which oracles do not exist and the correct program output is unknown, the results are not processable by human beings, or the implementation of such oracles is not practically feasible [Wey82]. Pseudo-oracles are independently implemented versions of the program, in the same way of Avizienis and Randell approaches, that achieve the same goal and result of the one under test. The approach can reveal errors when the computed outcomes of all independently versions of the program do not agree. Although the error cannot be confidently localized in the program under test, the validity of the result is at least questionable. The Pseudo-oracle mechanisms requires at least two versions of the same program to be implemented, and at least three to automate the process.

Exception Handling Exception handling is a classic mechanism to handle predefined classes of faults, by the mean of recovery actions coded in the system at design time. One of the first formalizations of exceptions was made by Goodenough [Goo75]. The author classifies the exceptions into four categories: a *range failure* occurs when an outcome of a procedure does not satisfy an output assertion. A failure on an input assertion is classified as *domain failure*. Exceptions are not only tied to system failures, indeed, the author proposes two classifications, *classify result* and *monitoring*, that are raised when the invoker needs more information on the result of the procedure or on the progress of the computation, and not when a failure occurs. Goodenough also proposes three strategies to handle exceptions. While handling an exception, we may require to terminate the procedure that raised the exception (*escape*), resume the procedure is at the end of the handler actions (*notify*), or allow both strategy (*signal*). Reasoning on the existing exception handling mechanisms, Liskov and Snyder propose a simpler model based on a single-level scope and that imposes the termination of the procedure [LS79]. Relying on the concepts of exception handling and Recovery Blocks, respectively by Goodenough and Randell, Cristian proposes a semantic and defines the usage of a notation that joins mechanisms of exception handling and fault tolerance [Cri82]. Lang *et al.* proposes a detailed survey on exception handling techniques [LS98].

Self-Checking Programming The idea of relying on software redundancy to increase reliability was discussed, in 1975, by Yau *et al.* who proposed the idea of self-checking software. This approach uses injected redundancy to automatically check the program behavior at runtime and verify the correctness of the running operations. The authors discuss the differences of designing self-checking systems by adding redundancy at different levels, such as at system, module, instructions or data level. The lower the level is, the finer the correction strategy that could be applied is [YC75]. Com-

bining Self-Checking Programming and N-Version Programming, Laprie *et al.* describe an approach called *N Self-Checking Programming* [LBK90]. Differently from N-Version Programming, where the components of the system that are developed in several variants are used as implicit oracles, in the N Self-Checking Programming approach, a component consists of either a variant and an acceptance test, used as explicit oracle, or two variants and a comparison algorithm. The variants run in parallel, and when the main one fails, a spare one can continue to deliver the service, and if the spare component fails, the main one can resume the execution again.

Robust Data Structures and Data Diversity A different set of strategies to improve reliability of software systems consists in increasing the dependability of data structures with data redundancy techniques. One of the first approaches has been proposed by Connect *et al.* in the early seventies [CPW72]. The authors propose to use software error detectors and program correctors, which they call *audits*. Audits use redundant software structures to detect and locate errors, and restore the memory to a consistent state to allow a correct execution of the system. Taylor *et al.* proposed data redundancy to increase the robustness of data structures, and increase the overall reliability of the system [TMB80]. They identify three common forms of data redundancy to make data structures more robust: the count of nodes in a data structure, identifier fields, and additional pointers. For example, the robustness of a simple linear list can be enhanced by adding identifier fields to each node, making the list circular by adding a pointer from the last node to the first one, and tracking the total count of the nodes. In this case an error can be easily detected by checking the consistency of this redundant information. Knight *et al.* analyze the failures caused by some particular regions of input, and propose a fault tolerant approach called *re-expressions* [AK88]. Data re-expression is the generation of logically equivalent data sets, that when used as inputs of an algorithm, produce the same results. The approach takes advantage of equivalent data input to overcome a fault caused by the original input data set. The authors implement data diversity borrowing the idea from the Recovery Blocks and N-Version Programming techniques. *Retry blocks* are blocks of code subjected to an acceptance test, and when the test fails a new set of input data is re-expressed and used to re-execute the block; *N-Copy Programming* consists in n parallel execution of the same program with n equivalent input data sets.

2.2 Self-Healing

The complexity of safety-critical systems challenged and is still challenging the researchers to propose and develop new fault tolerance approaches and techniques.

In the last fifteen years, the increase in the dependence of the society from software systems has emphasized the need of high reliability in a large class of systems where requirements are not safety-critical, but where failures may have big impact on

the human society, such as in business or social systems. For this class of software systems, the high costs of specialized hardware, multiple designs, or multiple versions development are not acceptable. Autonomic and self-managed systems have recently emerged as a possible way to cope efficiently with runtime problems. In these systems, the systems are enabled to cope with errors that are still in the code in the production environment, and sometimes only with specific classes of errors.

IBM proposed autonomic systems as a new development paradigm, where systems have knowledge of themselves and are aware of the environment in which they operate. Autonomic systems are able to reconfigure and optimize under unpredictable conditions, they can recover from and heal failures, as well as protect from external attacks [Hor01, KC03]. In the IBM vision, such systems are aware of the behavior of the components they are composed of, of their external interfaces, and how to compose these components to make them cooperate to achieve the self-management goal. Autonomic systems are usually coordinated by a control loop that automatically collects the details it needs from the system, analyzes these details to decide if a change is needed and, in that case, creates a plan of actions to react to the perturbations, and finally performs those actions [IBM06].

Kramer and Magee tackle the architectural challenge of self-managed systems with a three-layer reference model. The bottom layer is the controller of the components. It consists of a set of interconnected components that accomplish the application function of the system, sensors to interact with the environment, and a manager to support the planning to react to stimulus. The middle layer executes the plans received from the bottom level to handle new situations. The uppermost layer is the deliberation layer. This layer computes long term plans to achieve high-level goals, considering the current state of the system [KM07].

Self-managed systems deal with many properties. Here we focus on self-healing systems that automatically recover from functional failures. Self-healing systems can automatically detect failures, diagnosis the faults and propose one or more actions to heal the faults, or avoid the failures.

Force This set of healing approaches try to fix the fault by forcing the system to behave as expected, to reach the correct goal. Meyer *et al.* and Zeller *et al.* propose two techniques that derive bug fixes based on behavioral models. The former approach, implemented in a tool called PACHIKA [DZM09], extracts the execution traces of passing and failing test cases and builds behavioral models by mining these traces. The models are final state machines that abstract on the values of numbers and booleans. The approach detects anomalies by comparing passing and failing runs, and learns preconditions from the former, and violations of such preconditions from the latter. All the method calls that are relevant for the failure are selected and analyzed to discover precondition violations. If a method call violates at least a precondition, the tool generates a fix by deleting the violating call, or inserting the calls that make the state of

the system to agree with the model mined with the passing runs. In a further work, AutoFix-E [WPF+10], they rely on contracts and boolean query abstraction: contracts associate specifications and methods, boolean queries abstract the object state using boolean-valued functions with no arguments. As in the previous work, they use test cases to derive models that describe the system and the occurring failure. They extract the abstract object state model from the executions of the test cases using boolean queries, and profile a fault by comparing passing and failing states of test case executions and storing all the predicates that hold in the passing run but not in the failing ones. They derive finite state behavioral models of the classes under test from the error-free runs, and use the models to propose candidate fixes by determine sequences of calls which can change the object state appropriately.

Similarly, Perkins *et al.* propose ClearView [PKL+09]. The ClearView approach is based on invariant checking and fixing strategies to steer and force the system to agree to violated invariants. The tool uses Daikon [EPG+07] to observe the normal execution of the system to learn invariants, and it uses one or more detection mechanisms to catch violations of such invariants. The captured failures are then correlated to the relevant invariants by cross-checking those invariants that hold in the normal executions but not in the failing one, and the invariants that are violated before the failure occurs. They propose three fixing strategies: one that enforces the state of the system to hold the invariants correlated to the failure, one that forces the registers to hold the related invariants and one that changes the flow of the execution.

Yet another way to handle failures is to lead the program execution to avoid certain paths. Pagano *et al.* propose a tool, FastFix, that generates error reports to help developers reproduce and find the root cause of failures, and automatically generate patches to avoid certain types of failures. By analyzing the application code, the tool removes specific issues by applying a supervision mechanism proposed by Gaudin *et al.* to exclude the faulty execution paths [GVNH11]. In their work, Gaudin *et al.* apply the Supervisory Control Theory to produce a supervisor system that is embedded within the original system and that can steer the system to avoid failing situations. The supervisor uses a finite state machine that is built from the source code of the program and that represents the observable behavior of the program, it monitors the execution of the original program and when an unhandled failure is caught, it automatically synthesizes an execution path from the model to avoids the failure.

Guo proposes to convert unhandled runtime exceptions into special objects called *Not Available* objects (NA), that represent the exceptions [Guo11]. When an expression throws an uncaught exception, a NA object is created and stored in to the expression target in place of the missing result, a log file is populated with the details of the failure, and the program execution continues. If an NA object is involved in a computation, special rules are followed to compute the result, for example a binary operation involving an NA object will return the NA object itself. With this approach, long running programs or scripts can complete their executions and produce partial results and log

files related to the failures, instead of quitting with no results.

Lightweight variants of N-Version Programming The underlying idea of N-Version Programming has been recently exploited by other researchers for different kinds of software systems.

Liskov *et al.* propose to mask software errors with process replicas [CRL03]. The technique, called BASE, implements a replication technique where the implementation details of the replicas are abstracted, to enable the reuse of off-the-shelf software. The N-Version Programming idea of implementing several different versions, of the software, is here exploited by using off-the-shelf software, to decrease the development costs. Each different version runs in a different replica, and the abstraction layer masks the discrepancies among the versions, while data consistency and determinism are guarantee by *conformance wrappers*.

Gashi *et al.* propose a technique to increase the dependability of SQL servers by taking advantage of the well defined interfaces of such databases and relying on the parallel execution of different implementations of SQL databases [GPSS04]. The database variants are executed in parallel and a voting algorithm determines the final result of the queries in case of failure of one of them. A middleware guarantees a deterministic behavior and data consistency among the databases.

Hoser and Cadar [HC13] tackle the problem of the introduction of new faults in software upgrades. The authors propose to concurrently use two versions of the software, the latest release and an old one. The two versions run synchronously and in parallel; whenever one of the two crashes, the code of the the other version is used to survive to the failure.

The N-Version Programming model has been exploited to increase the dependability of Web services. Looker *et al.* [LMX05] and Dobson [Dob06] propose to rely on different equivalent versions of services running in parallel, as in the original idea of Avizienis, that can be developed singularly or reused as off-the-shelf services to decrease the costs, coupled with a voting system that accepts the majority of the results as correct outcome.

Lightweight Recovery Blocks Recovery Blocks have been adapted and used in different domains. The variants of the original approach rely on the trend of new development approaches to design modular systems. Randall's Recovery Blocks are seen as modules, that can be used as off-the-shelf components, often released by different organizations, to gain a sufficient level of reliability and keep the cost of development lower.

Cabral *et al.* augment classic exceptions handing technique with a dynamic selection of Recovery Blocks as recovery actions based on the failures. The main idea is to provide several recovery actions and, when the system fails and an exception is caught, select the most suitable action to cope with the type of exception [CM11].

Similarly, Demsky *et al.* propose an approach that mitigates failures that break data dependencies at runtime [DZM10]. The approach relies on recovery tasks, tasks written by the developers that can operate and achieve partial results even if a failure in the system computation causes some of the parameters to be unavailable. The approach characterizes the correct behavior of the system with a static analysis based on an Abstract State Transition graph. When a failure is reported, a recovery algorithm uses the static analysis to determine the possible intended execution and computes a sequence of recovery tasks to achieve the intended goal. The recovery tasks can mark the parameters that are not available due to a failure as *failed*, and excludes them from the computation [DD08].

Samimi *et al.* propose to use formal specifications to avoid contract violations at runtime [SAM10]. When a contract is violated, the approach, called *Plan B*, uses a constraint solver to analyze the specification of the failed code, starting from the same dynamic program state, and produce the expected result. The authors provide a language to let the developers write the specifications that can be executed and that can be incorporated into the source code.

Recovery Blocks for the Web The natural architectural predisposition of Web services as composition of several services, makes the Randell's Recovery Blocks approach particularly appealing in this domain. Many organizations offer services on the Web, many of these services are similar or equivalent in terms of functionality and quality of service offered. Services can be considered as Recovery Blocks that are designed and developed independently.

The Dobson's approach relies on a Recovery Blocks strategy. The developer can decide what strategy to apply when a service fails: the parallel execution of the services can be replaced with a *retry* strategy that sequentially executes different equivalent services [Dob06].

Many authors propose to extend BPEL processes with self-healing capabilities to monitor the behavior of Web applications and, with a reactive architecture, to select an adequate strategy to cope with anomalous behaviors. Web services can be used as off-the-shelf components, developed by different organizations but that, often they offer the same functionality. A typical strategy consists in replacing a service invocation with an invocation of a different, functionally equivalent service [MMP06, TBFM06, BG07, BGP07, MB08, STN+08].

Workaround Redundant code has been proved to be effective in avoiding failure in Web applications. Carzaniga *et al.* studied the feasibility of using the intrinsic redundancy of software systems to overcome faults [CGP08b, CGP08a, CDP+09] and prove its effectiveness in stateless Web applications [CGPP10a, CGPP10b]. In this thesis, we propose a technique to automatically generate workarounds for stateful applications by exploiting the intrinsic redundancy of software systems, thus properly

extending the seminal work by Carzaniga *et al.*.

At a lower level, Engel *et al.*, propose a toolkit for weaving aspects into operating system kernel, implemented as dynamically exchangeable kernel modules. The tool provides before, after and around point-cuts. As an example of application of the approach, they illustrate a self-healing system that automatically adds new virtual memory space if required [EF05].

Wrappers Component-based systems heavily rely on re-use, but sometime adapting components to integrate them into new systems can be impossible or expensive. Wrappers are specialized components inserted as bridges in between the new system and other components, that help the integration by dealing with the communication and the control flows.

The work of Popov *et al.* focuses on protective wrappers [PRRS01]. In this work, wrappers are designed to improve the dependability of a software system by protecting the system from anomalous behaviors of off-the-shelf components and integration problems. The developers of the system have to design and implement the wrappers, which include the acceptable behavior of the integration of the system under control and the component. The developers also have to specify how to tolerate or mitigate the effects of the detected failures in the boundaries between the system and the components.

Similarly, Chang *et al.* propose a methodology and a technique based on healing wrappers to reduce the failure occurrence caused by integration problems [CMP09]. In this work, unlike the one proposed by Popov *et al.*, wrappers are written by the developers of the components and include healing strategies for common misuses and failures raised by the component itself. Later, the system developers who use such components, can inject these wrappers into their system and take advantage of runtime healing of failures provided by the components. Wrappers are composed of a detection mechanism, one or more healing strategies, and one or more injection points. When the wrapper catches a failure, the state of the system is analyzed to understand if the failure has been raised by the component, and an healing strategy is invoked to solve the problem.

Denaro *et al.* show an approach to develop adaptation strategies that aim to improve the service interchangeability for service-oriented applications based on standard APIs [DPT13]. The approach, called Test-and-Adapt, consists in a set of test and adaptation plans. Each plan consists of a set of parametric test cases and an associated set of parametric adaptors. The test cases identify inconsistencies between the applications and the APIs and trigger suitable adaptors to solve the identified inconsistencies.

Evolution Genetic programming is a methodology to solve problems inspired by natural evolution. Some self-healing approaches exploit genetic programming to derive, from a failing program, a correct program that fixes the fault. Genetic programming

uses the intrinsic redundancy to produce variants of a failing program, and selects the variants that appear to be "more" correct. Such variants are progressively refined and evolved in a process of mutations and recombinations, until a solution, or an acceptable approximation of it, has been reached. To evaluate whether a variant is evolving towards an acceptable solution or must be discarded, each variant is measured with a *fitness function*. A fitness function determines if a variant "behaves well" or not. Those variants that behave well are chosen to breed, and to evolve into a new set of programs that are, hopefully, a step closer to the solution. There are two main strategies to create new programs, called genetic operations: *crossover* and *mutation*. The former creates a child by combining parts of code of two selected programs, the latter, by choosing a code fragment of a program and altering it following a set of predefined rules.

Researchers applied genetic programming in the context of automating fault fixing. Debroy *et al.* propose to use Tarantula [JH05] to localize and rank all the possible locations of the fault in the code. Then, starting from the most highly ranked location, they evolve the program and produce a set of variants. To evolve the program they use a predefined set of mutation primitives, such as replacement of arithmetic and logic operators and negation of decision making statements. When a variant is produced, it is compared on the fitness function. The authors use the original test suite to define the fitness function: they keep a variant if it passes all the test cases already passed with the previous generation; in other words, a variant is killed if it fails more test cases than its originator. The iterative evolution stops upon a certain threshold based on the effort to produce a better variant for a certain fault location in the fault ranking [DW10].

Arcuri *et al.* propose a more sophisticated approach that combines genetic programming, search-based testing and test co-evolution [AY08]. They propose a richer set of primitives to produce the variants, compared to Debroy *et al.*, such as mutation of arithmetic and boolean expressions, of control flow statements, of variables and arrays. They define the fitness function as the distance of the output of the computation of a variant from the expected result. The value of the fitness function increases in proportion to the difference of the output from the expected result. The goal of this algorithm is to minimize the value of the fitness function. To decrease the computational costs of running the test suite to evaluate the variants, the authors use a search-based testing approach to try to find at least a test case that fails on the variant. To increase the chances to find more faults, they also evolve the test suite to create a competitive co-evolution between the program and the test suite.

Another genetic programming approach is proposed by Weimer *et al* [WNLGF09]. Similarly to Arcuri *et al.*, test cases are used to check the program functionality and correctness. This technique uses a different genetic programming algorithm: new evolutionary programs are produced by changes based on structures already present in other parts of the program. Changes are also applied only on those regions of the program that are relevant to the error. Those regions consist of paths, that are weighted based on occurrences in failing and passing test cases. The fitness function is a weighted sum

of the positive and negative executed test cases produced by an evolutionary program. The relative fitness in the population is used to select those individuals that are candidates to propagate the evolution. The evolution is done by mutation or by crossover. The mutation consists of deletion, insertion, or swapping of statements. The crossover considers only those statements that are visited while executing the test cases. In a further work, Weimer *et al.* [LGDVFW12], enhance the approach by adding Tarantula [JH05] as fault localization mechanism, and by calculating an approximation of the fitness function to minimize the number of test cases required.

Environmental Changes Long-running and computation-intensive applications suffer particularly from *age*. Non-deterministic faults can make the system crash, but often degrade the system performance because of memory leaks, memory caching, weak memory reuse, etc, and lead to system failures only after a long time [HKKF95]. Researchers proposed several approaches to tackle this situation.

Software rejuvenation periodically restarts an application to clean the environment and re-initialize the memory and the data structures, thus preventing the system to fail because of age [HKKF95]. The runtime costs of rejuvenation may be high. For example restarting a Web server may result in a service downtime, or restarting a transactional system may cause losses of the current transaction incurring in rollback costs [GT07]. Thus, rejuvenating the system at the right time is important. A strategy to select the optimal interval is to measure some system attributes that show the symptoms of aging [GLVT06].

Similarly to rejuvenation, *micro-rebooting* reboots only the system components that failed or show aging signs [CKF+04][CCF+02].

A different approach consists of re-executing failing programs under a modified environment. Qin *et al.* propose a technique called Rx that dynamically changes the environment based on the failure symptoms, and re-executes the faulty program in the modified environment. They propose many environment changes, for instance memory management strategies, process re-scheduling, user request dropping [QTSZ05].

Checkpoint and Recovery Rollback mechanisms are often employed to safely recover and re-execute at runtime failing tasks. Elnozahy *et al.* propose a simple approach. Upon a failure, they suggest to rollback the system to a previously saved checkpoint and re-execute the failed code to fix temporary problems that may be related to the particular environment or synchronization conditions [EAWJ02].

Checkpoint and Recovery approaches are also used to increase the performance of rejuvenation. Wang *et al.* couple a checkpoint mechanism with memory rejuvenation to prevent problems related to memory leaks [WHV+95]. Gang *et al.* start with the observation that a checkpointing system reduces the completion time of programs that fail, and show that it is possible to obtain further time reduction by adding rejuvenation [GHKT96].

Self-healing by design Some authors propose techniques to produce systems that are specifically designed to have self-healing capabilities. Breitgand *et al.* propose a framework called PANACEA [BGH+07]. The framework includes a methodology to steer the design and the development of a system to include a set of healing components that will be used at runtime. Such components are in charge of monitoring and reconfiguring the system in case of misbehaviors. The healing components can be application-specific, when they contain some specific knowledge of the system, or generic, so that they can be reused across different systems. They interface with the system through code annotations that enable healing components on specific parts of the system.

Shehory proposes another framework to integrate self-healing capabilities into a system called SHADOWS [She08]. To enable self-healing capabilities in a component of the system, a model of its behavior must be included at design time. The behavioral model enables the runtime monitoring capability and, upon a violation of such model, a specific healing action can be opportunistically taken.

Another model-driven approach is proposed by Wei *et al.* [WYCL11]. The idea is to have two separated models, one for the functional and non-functional requirements of the system, and another one for the self-healing capabilities. The self-healing model includes a model of the possible classes of defects of the system and their consequences, and also a model that describes the healing strategies to deal with those classes of defects. Then, the self-healing model is merged into the functional model using a model composition strategy, for instance using a weaving model to link the two models together.

In this chapter we have seen how redundancy has been exploited to increase the reliability of software systems. We have seen that introducing redundancy deliberately is expensive and it requires additional steps in the design and the implementation of the systems. We have also seen that recently, researchers proposed methodologies and techniques that try to relax these requirements and that deal with faults at runtime by reasoning on the failure and rather that relying on multiple implementation of the critical parts of the system.

Although researchers proposed autonomic systems as a way to conveniently exploit some knowledge of the system, the production or the gathering of such knowledge still requires a lot of effort. Moreover, many of the techniques described in this chapter face the fault offline and propose a patch, thus they require the system to stop. In this context, the open challenge is to propose a methodology that can automatically avoid failures at runtime at low cost and with little effort from the developers.

Chapter 3

Automatic Workarounds

> *System failures might be avoided by means of workarounds. Workarounds are operations that provide the same functionality of a failing operation, but do not suffer from the same failure. Specifically, workarounds are operations that are observationally equivalent in their intended effect with respect to some original operations, but when the original operation fails, they do not fail. We propose to avoid runtime system failures by automatically apply workarounds to a failing code.*

In this chapter we introduce the concept of software redundancy. We argue that developers release software systems with a large amount of redundancy. We propose to exploit software redundancy to avoid system failures at runtime, and we define an approach, Java Automatic Workaround (JAW), to do it automatically.

Modern software systems, especially modular systems and reusable components, are designed to be easily adopted by a broad variety of applications. Therefore, they provide interfaces with many variants of the same functionality, and many functionality are accomplished in different ways. Such variants are often similar enough in their semantics to be easily exchangeable, and frequently differ in their implementation. For instance, let's consider container libraries. Typically, they offer similar operations at the interface level, such as *add(element)* and *addAll(collection)*, which can be interchanged with few adaptations.

Let us take, for instance, the implementation of the ArrayList class (in *Java 7*) that exposes the two methods add(E e) and addAll(Collection<? extends E> c) (Listing 3.1). The two methods are clearly implemented differently and are easily interchangeable with with few adaptations.

Sorting libraries offer another similar example. The Arrays class exposes the method sort, which is overloaded with different versions, such as sort(int[] a) and sort([Object[] a) (Listing 3.2). The two different versions implement different sorting algorithms: the former uses a tuned quicksort, while the latter a modified mergesort and timsort.

This form of software redundancy that we call *intrinsic* redundancy is often already

```
1  public boolean add(E e) {
2    ensureCapacityInternal(size + 1);
3    elementData[size++] = e;
4    return true;
5  }
6
7  public boolean addAll(Collection<? extends E> c) {
8    Object[] a = c.toArray();
9    int numNew = a.length;
10   ensureCapacityInternal(size + numNew);
11   System.arraycopy(a, 0, elementData, size, numNew);
12   size += numNew;
13   return numNew != 0;
14 }
```

Listing 3.1. Methods add and addAll exposed by ArrayList

```
1  public static void sort(int[] a) {
2    DualPivotQuicksort.sort(a);
3  }
4
5  public static void sort(Object[] a) {
6    if (LegacyMergeSort.userRequested)
7      legacyMergeSort(a);
8    else
9      ComparableTimSort.sort(a);
10 }
```

Listing 3.2. Sorting algorithms implemented with quicksort, mergesort and timsort in the Java Arrays class

present in modular software systems and reusable components. In this thesis we show that we can use the intrinsic software redundancy to increase the reliability of software systems in a cost effective manner. In Chapters 4 we present a qualitative and a quantitative analysis to show how intrinsic redundancy is common in modern software systems, focusing on Java libraries. In Chapter 8 we evaluate the effectiveness of the intrinsic redundancy we found in such Java libraries by means of a tool we developed to add self-healing capabilities to software systems that exploit the intrinsic redundancy.

We propose an approach applicable to a broad variety of general-purpose and possibly long-running systems that is able to overcome a wide range of failures, and that incurs low costs. JAW automatically recovers software systems from failures and relies on the intrinsic redundancy that software systems offer to automatically avoid system failures at runtime. With several limitations, the approach has been previously implemented into a prototype and applied in the Web domain to show its effectiveness. In this thesis we propose a general and reusable framework to automatically deploy effective workarounds.

In the following sections we introduce the concept of the intrinsic redundancy with a case study based on the Java library SWT, a graphical library provided by the Eclipse Foundation (Section 3.1). We then introduce our approach (Section 3.2), and finally, we compare it with prior results in the Web domain (Section 3.3).

3.1 SWT Library Case Study

In this section we present a case study, the SWT library, to show the presence of redundancy in software systems. We use the same case study in the next sections to demonstrate that such redundancy can be used to overcome failures.

SWT[1] (Standard Widget Toolkit) is a Java graphical widget toolkit developed by the Eclipse Foundation that provides access to the operating systems user-interface facilities. SWT accesses the operating system native GUI, and provides a portable Java framework, which is unique for each platform. The library is designed to provide the developers with a wide range of graphical entities to draw GUIs, such as buttons, labels, lists, menus, tables, bars, and so on. Each graphical widget is provided with a full set of methods to create, customize, and manage widgets.

The documentation of the widgets witnesses the use of several programming practices in the design of the class interfaces, such as method overloading and overriding. Method overloading is used to provide multiple behaviors of the same method, by allowing the same method to have different input and output types. Method overriding is used to implement specialized behavior of the inherited methods. Such object-oriented programming practices foster the proliferation of semantically equivalent code vari-

[1] `http://www.eclipse.org/swt`

ants. For instance, in the class *Table*, for the method *setSelection*, there exist five different variants to select items: *setSelection(int index)* and *setSelection(int[] indices)* select an item based on either an index or an array of indexes of the items to select, respectively; *setSelection(TableItem item)* and *setSelection(TableItem[] items)* select an item based on item or the items themselves to select, *setSelection(int start, int end)* selects items based on the range of indexes of the items to select. These five variants are semantically equivalent, and can be exchanged with simple adaptations in the parameters.

Another common practice is to provide methods that interact with objects at different granularity. For instance, it is common to provide pairs of methods such as *select(item)* and *selectAll()*, *clear(item)* and *clearAll()*, *getItem(item)* and *getItems(items)*. In most of the cases it is straightforward to replace one method, such as *clearAll*, with a loop of calls to the pairing method, in this case *clear*. This practice leads to a proliferation of methods that can be easily adapted to behave exactly in the same way.

There are also more complex forms of redundancy. Let us consider the interface of the class *TreeItem* of the SWT library. A TreeItem is *a selectable user interface object that represents a hierarchy of tree items in a tree*, that is, an object Tree can contain many TreeItem objects, and every object TreeItem can contain many TreeItem objects. To remove all the children of a TreeItem, the interface exposes the method *removeAll()*, which *removes all of the items from the receiver*, and the method *setItemCount(int count)*, which *sets the number of child items contained in the receiver*. The two methods *removeAll()* and *setItemCount(int count)* are indistinguishable when the parameter *count* of setItemCount is set to zero. The interface of the class TreeItem also exposes the method *dispose()*, it *disposes of the operating system resources associated with the receiver and all its descendants*, and can also be used to remove all the children of a TreeItem object. Intuitively, the three methods present the same behavior and they can be interchanged to obtain the same result, that is to remove all the children of a TreeItem object. They provide three variants for the same functionality, and thus they are redundant.

Assuming that one of three methods previously introduced fails, we can rely on the knowledge about the redundant methods to avoid the failure, by replacing the failing method with a redundant one. The next paragraph shows a case study of a fault that can be avoided by using intrinsic redundancy in the SWT library.

The removeAll() method issue. The method *removeAll()* of the SWT implementation for the Macintosh platform contains a fault[2] in the version 3.5 of the library. The method, which should remove all the children of a given TreeItem object, also removes its parent TreeItem object and all the siblings of the given TreeItem object, that is clearing a misbehave. In the following part of this section, we show how this fault affects the program execution though a simple example.

[2] `https://bugs.eclipse.org/bugs/show_bug.cgi?id=279313`

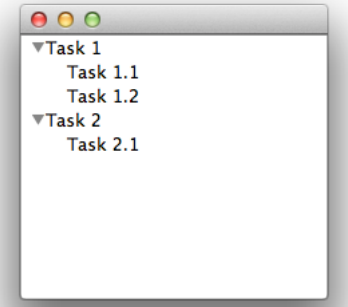
Listing 3.3 is a program that, using SWT, creates a window (lines 3-4) that contains and displays a tree structure (line 5). The program, then, adds two main TreeItem objects to the tree, two children to the first item, and a child to the second one (lines 7-17). Finally, it opens and displays the window (line 19). Figure 3.1(a) shows the result produced by Listing 3.3. The window contains a tree with two items, and each of them contain, respectively, two and one children.

```
1  public class Example {
2    public static void main(String[] args) {
3      Display display= new Display();
4      Shell shell = new Shell(display);
5      Tree tree = new Tree(shell, SWT.SINGLE | SWT.BORDER);
6
7      TreeItem item1 = new TreeItem(tree, SWT.NONE, 0);
8      item1.setText("Task 1");
9      TreeItem item1a = new TreeItem(item1, SWT.NONE, 0);
10     item1a.setText("Task 1.1");
11     TreeItem item1b = new TreeItem(item1, SWT.NONE, 1);
12     item1b.setText("Task 1.2");
13
14     TreeItem item2 = new TreeItem(tree, SWT.NONE, 1);
15     item2.setText("Task 2");
16     TreeItem item2a = new TreeItem(item2, SWT.NONE, 0);
17     item2a.setText("Task 2.1");
18
19     shell.open();
20   }
21 }
```

Listing 3.3. Simple program that draws a tree of items within a window

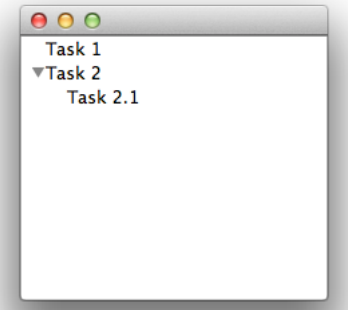
Let us now assume that we want to remove all the children of the first TreeItem, called *Task 1*, before displaying the window. For this purpose we can use the method *removeAll()* exposed by the class *TreeItem*, by calling it before the windows is displayed, as Listing 3.4 shows. The method *removeAll()* is invoked at line 11. The intended behavior of Listing 3.4 is to display a window containing a tree structure with a top level item, *Task 1*, childless, and a second top level item, *Task 2*, with one child. Figure 3.1(b) shows the actual result of Listing 3.4: an empty window, which is clearly different from the expected result.

We inspected the documentation of the interface of the class *TreeItem*, and found that there are three equivalent ways to remove all the children of a given item: the methods *removeAll()*, *setItemCount(int count)*, and *dispose()*. These three methods are equivalent in their expected behavior, so that it should be possible to exchange them to obtain the same result. This means that we can replace method *removeAll()* with *setItemCount(int count)* in the line 11 of Listing 3.4. The method *setItemCount(int*



(a) Result of the program in Listing 3.3: before the children of *Task 1* are removed

(b) Result of the program in Listing 3.4: the entire tree structure is removed



(c) Result of the programs in Listing 3.5 and 3.6: the workarounds produce the expected result

Figure 3.1. The tree structure produced by the codes in the example

```
1  public class Failure {
2    public static void main(String[] args) {
3      Display display= new Display();
4      Shell shell = new Shell(display);
5      Tree tree = new Tree(shell, SWT.SINGLE | SWT.BORDER);
6
7      //Operations to populate the tree
8      [...]
9
10     //Failing call:
11     item1.removeAll();
12
13     shell.open();
14   }
15 }
```

Listing 3.4. Program that fails to remove only some items

count) requires a parameter that indicates how many children the caller item will have after the call to such method. We set the parameter to zero to achieve the same effect of method *removeAll()*. The resulting code is presented in Listing 3.5, with the *item1.setItemCount(0)* statement at line 11. The results of executing Listing 3.5 is shown in Figure 3.1(c) that correctly shows the tree structure as required. The method *setItemCount(int count)* does not suffer from the same fault of the method *removeAll()*, and can avoid the failure.

We can repeat the same experiment by replacing the faulty method *removeAll()* with the redundant variant *dispose()*, also exposed by the class TreeItem. Such method disposes both the caller object and all its descendants, and thus, to preserve the caller object itself, that in this case is the TreeItem object named *Task 1*, we have to apply the *dispose()* method only on its descendants. We can loop on all the children of the TreeItem *Task 1*, and for each of them, call the method *dispose()*. For this purpose, we used a *for* loop, as showed at line 11 to 13 of Listing 3.6. The results of executing Listing 3.6 is shown in Figure 3.1(c).

This experiment shows that we can, indeed, rely on the redundancy to avoid a failure. When a redundant piece of code avoids the failure, we call it a workaround. In this particular example, we could find two workarounds by looking at the interface of the class. Intuitively, there is not guarantee that redundancy at the interface level reflects redundancy at the code level. If the redundant methods share exactly the same code, in the same environmental conditions, they will follow the same execution and they will fail in the same way. In our example, what made the two alternative methods succeed is that the they execute different code than the original failing method.

Let us have a look at the source code of the three methods and at their execution trace. The Listing 3.7 shows the source code of the *revoveAll()* method that, to recall,

```
1  public class Workaround1 {
2    public static void main(String[] args) {
3      Display display= new Display();
4      Shell shell = new Shell(display);
5      Tree tree = new Tree(shell, SWT.SINGLE | SWT.BORDER);
6
7      //Operations to populate the tree
8      [...]
9
10     //Workaround:
11     item1.setItemCount(0);
12
13     shell.open();
14   }
15 }
```

Listing 3.5. Program that uses *setItemCount* as workaround

```
1  public class Workaround2 {
2    public static void main(String[] args) {
3      Display display= new Display();
4      Shell shell = new Shell(display);
5      Tree tree = new Tree(shell, SWT.SINGLE | SWT.BORDER);
6
7      //Operations to populate the tree
8      [...]
9
10     //Workaround:
11     for (TreeItem treeItem : item1.getItems()) {
12       treeItem.dispose();
13     }
14
15     shell.open();
16   }
17 }
```

Listing 3.6. Program that uses *dispose* as workaround

fails in removing all the children of a given item on the three. At line 4, the method removeAll() makes a call to the method *setItemCount(int count)*, whose source code is shown in Listing 3.9. At runtime, the caller of the *revoveAll()* method is an item at the top level of the tree, in particular it is the object *item1* labeled as "Task 1". Accordingly to the code of the method *revoveAll()*, the inner method *setItemCount(int count)* is invoked on its parent, which is the tree itself. This means that the children of the tree object are set to zero and thus the whole tree is removed.

```
1 // org.eclipse.swt.widgets.TreeItem.removeAll()
2 public void removeAll () {
3   checkWidget ();
4   parent.setItemCount (0);
5 }
```

Listing 3.7. Source code of the method *removeAll*

This behavior is more clear by looking at the execution trace of the failing program. Listing 3.8 shows a slice section of the execution trace related to the failing statement *item1.removeAll()* (at line 11 in Listing 3.4) in the failing program. In the trace, we can identify the sequence

```
TreeItem.release(boolean)
TreeItem.releaseChildren(boolean)
```

which draws out the objects. Such sequence is recurring five times in the trace (lines 13-14, 15-16, 21-22, 31-32, and 33-34). Since we have five items on the tree, two top level items with respectively two and one children, this means that all the items in the tree are removed, leading to the wrong result.

The first workaround that is proposed in the Listing 3.5 contains the statement *item1.setItemCount(0)*, instead of *item1.removeAll()*. The source code of the method setItemCount is shown in Listing 3.9. In this case, the object *item1*, labeled as "Task 1", calls the method *setItemCount(int count)*. As seen before, *setItemCount(int count)* removes all the children of the caller object, so only the two children of "Task 1". This behavior is confirmed by the execution trace shown in Listing 3.10. Indeed, the sequence release/releaseChildren is recurring only twice (lines 12-13, and 18-19), which means that only the two children of "Task 1" are removed.

A similar reasoning can be applied to the workaround proposed in the program in Listing 3.6. The method *dispose()* is an inherited method, an its source code is shown in Listing 3.11. It accesses directly the operating system facilities to draw out the items from the tree, and in its execution trace (Listing 3.12) it is still possible to see the pattern release/releaseChildren recurring twice (lines 4-7, and 26-29).

As we have shown in the example described in this section, the three methods *removeAll()*, *setItemCount(int count)*, and *dispose()*, are thus not only redundant at the

```
1  TreeItem.removeAll()
2  Tree.setItemCount(int)
3  Tree.checkItems()
4  Tree.setItemCount(org.eclipse.swt.widgets.TreeItem, int)
5  Tree.getItemCount(org.eclipse.swt.widgets.TreeItem)
6  TreeItem.clearSelection()
7  TreeItem.getExpanded()
8  TreeItem.clearSelection()
9  TreeItem.getExpanded()
10 Tree.getSelection()
11 Tree.outlineView_numberOfChildrenOfItem(long, long, long, long)
12 Tree.selectItems(org.eclipse.swt.widgets.TreeItem[], boolean)
13 TreeItem.release(boolean)
14 TreeItem.releaseChildren(boolean)
15 TreeItem.release(boolean)
16 TreeItem.releaseChildren(boolean)
17 TreeItem.releaseWidget()
18 Item.releaseWidget()
19 TreeItem.deregister()
20 TreeItem.releaseHandle()
21 TreeItem.release(boolean)
22 TreeItem.releaseChildren(boolean)
23 TreeItem.releaseWidget()
24 Item.releaseWidget()
25 TreeItem.deregister()
26 TreeItem.releaseHandle()
27 TreeItem.releaseWidget()
28 Item.releaseWidget()
29 TreeItem.deregister()
30 TreeItem.releaseHandle()
31 TreeItem.release(boolean)
32 TreeItem.releaseChildren(boolean)
33 TreeItem.release(boolean)
34 TreeItem.releaseChildren(boolean)
35 TreeItem.releaseWidget()
36 Item.releaseWidget()
37 TreeItem.deregister()
38 TreeItem.releaseHandle()
39 TreeItem.releaseWidget()
40 Item.releaseWidget()
41 TreeItem.deregister()
42 TreeItem.releaseHandle()
```

Listing 3.8. Execution trace of the Listing 3.4

```
1  // org.eclipse.swt.widgets.Tree.setItemCount(int)
2  public void setItemCount (int count) {
3    checkWidget ();
4    count = Math.max (0, count);
5    parent.setItemCount (this, count);
6  }
```

Listing 3.9. Source code of the method *setItemCount*

```
1  TreeItem.setItemCount(int)
2  Tree.setItemCount(org.eclipse.swt.widgets.TreeItem, int)
3  Tree.getItemCount(org.eclipse.swt.widgets.TreeItem)
4  TreeItem.getExpanded()
5  Tree.getSelection()
6  Tree.outlineView_isItemExpandable(long, long, long, long)
7  Tree.outlineView_isItemExpandable(long, long, long, long)
8  Tree.outlineView_child_ofItem(long, long, long, long, long)
9  Tree._getItem(org.eclipse.swt.widgets.TreeItem, int, boolean)
10 Tree.outlineView_isItemExpandable(long, long, long, long)
11 Tree.selectItems(org.eclipse.swt.widgets.TreeItem[], boolean)
12 TreeItem.release(boolean)
13 TreeItem.releaseChildren(boolean)
14 TreeItem.releaseWidget()
15 Item.releaseWidget()
16 TreeItem.deregister()
17 TreeItem.releaseHandle()
18 TreeItem.release(boolean)
19 TreeItem.releaseChildren(boolean)
20 TreeItem.releaseWidget()
21 Item.releaseWidget()
22 TreeItem.deregister()
23 TreeItem.releaseHandle()
```

Listing 3.10. Execution trace of the Listing 3.5

```
1  // org.eclipse.swt.widgets.Widget.dispose()
2  public void dispose () {
3    if (isDisposed ()) return;
4    if (!isValidThread ()) error (SWT.ERROR_THREAD_INVALID_ACCESS);
5    release (true);
6  }
```

Listing 3.11. Source code of the method *dispose*

```
1  TreeItem.getItems()
2  Tree.checkData(org.eclipse.swt.widgets.TreeItem)
3  Widget.dispose()
4  TreeItem.release(boolean)
5  TreeItem.clearSelection()
6  TreeItem.getExpanded()
7  TreeItem.releaseChildren(boolean)
8  TreeItem.releaseWidget()
9  Item.releaseWidget()
10 TreeItem.deregister()
11 TreeItem.destroyWidget()
12 Tree.destroyItem(org.eclipse.swt.widgets.TreeItem)
13 Tree.reloadItem(org.eclipse.swt.widgets.TreeItem, boolean)
14 Tree.getSelection()
15 Tree.outlineView_isItemExpandable(long, long, long, long)
16 Tree.outlineView_isItemExpandable(long, long, long, long)
17 Tree.selectItems(org.eclipse.swt.widgets.TreeItem[], boolean)
18 Tree.setScrollWidth()
19 Tree.setScrollWidth(boolean, org.eclipse.swt.widgets.TreeItem[], boolean)
20 TreeItem.calculateWidth(int, org.eclipse.swt.graphics.GC)
21 TreeItem.getExpanded()
22 TreeItem.calculateWidth(int, org.eclipse.swt.graphics.GC)
23 TreeItem.getExpanded()
24 TreeItem.releaseHandle()
25 Widget.dispose()
26 TreeItem.release(boolean)
27 TreeItem.clearSelection()
28 TreeItem.getExpanded()
29 TreeItem.releaseChildren(boolean)
30 TreeItem.releaseWidget()
31 Item.releaseWidget()
32 TreeItem.deregister()
33 TreeItem.destroyWidget()
34 Tree.destroyItem(org.eclipse.swt.widgets.TreeItem)
35 Tree.reloadItem(org.eclipse.swt.widgets.TreeItem, boolean)
36 Tree.getSelection()
37 Tree.outlineView_isItemExpandable(long, long, long, long)
38 Tree.outlineView_child_ofItem(long, long, long, long, long)
39 Tree.outlineView_isItemExpandable(long, long, long, long)
40 Tree.selectItems(org.eclipse.swt.widgets.TreeItem[], boolean)
41 Tree.setScrollWidth()
42 Tree.setScrollWidth(boolean, org.eclipse.swt.widgets.TreeItem[], boolean)
43 TreeItem.calculateWidth(int, org.eclipse.swt.graphics.GC)
44 TreeItem.getExpanded()
45 TreeItem.calculateWidth(int, org.eclipse.swt.graphics.GC)
46 TreeItem.getExpanded()
47 TreeItem.releaseHandle()
```

Listing 3.12. Execution trace of the Listing 3.6

interface level, since it is possible to exchange them with no change in the semantics of the program, but they also execute some different code to achieve the same result. The developers of the library wrote these three methods to provide a more flexible and reusable interface, not for fault tolerance purpose or as a fix for the failure produced by the *removeAll()* method. This aspect suggests and supports our intuition that modern software systems are intrinsically redundant, and that we can exploit redundant code written for flexibility purpose, to achieve better reliability at a low cost.

This thesis intends to understand if software systems are intrinsically redundant, how to extract the redundancy, and how to use it automatically to avoid failures at runtime. In the next section we introduce our idea about how to recover a system from a failure and how to exploit intrinsic redundancy automatically.

3.2 Approach: Java Automatic Workaround

Our approach, JAW, automatically recovers a system upon a failure, changes the code accordingly to a given redundancy knowledge about the system itself, and re-executes the failed portion of code. This section shows how the information about intrinsic redundancy is expressed and how this information is used to enforce the system reliability at runtime.

The intrinsic redundancy that we exploit is expressed in a suitable language that we illustrate below. The effort to capture the redundancy depends on the knowledge of the system. The developers of the system itself have a deep understanding of the system and they know exactly how it behaves, so the effort to capture the redundancy is small. The users of the system, for example third-party components or libraries, may need to carefully investigate the interfaces, even though, as shown in our experiments, the effort is still limited: bachelor and master students can easily inspect the documentation of the systems for the first time and extract a large amount of intrinsic redundancy in few days. Collecting the intrinsic redundancy is a conceptual work that relies on a systematic reasoning on the functionality that the system exposes and their similarities, and it is achieved, for example, by counting of the expertize on the system or by inspecting the interfaces. The activity of gathering the intrinsic redundancy is different than the bug fixing activity because it does not include debugging or code inspection.

When captured, the intrinsic redundancy must be expressed in a formal way to make it usable in our approach. For instance, in the previous section we captured the knowledge that three methods, *removeAll()*, *setItemCount(int count)*, and *dispose()*, are equivalent, and also that *setItemCount(int count)* is equivalent to the other two when its parameters is set to zero. To express this knowledge we need to formalize several concepts:

- *removeAll()*, *setItemCount(int count)*, and *dispose()* are methods of *TreeItem*;

- the original method and its redundant method must be called by the same object;
- the parameter of the method *setItemCount(int count)* must be set to zero;
- all of the three methods are equivalent to each other.

We express these concepts by means of *rewriting rules*. A rewriting rule substitutes a code fragment with a different equivalent one, that is a fragment with the same observable behavior of the original one. Given the former example, the rewriting rules in Listing 3.13 capture and express all the knowledge. The symbol $\equiv$ means that the two statements are equivalent, that is the first statements can be replaced by the second, and vice-versa; the symbol $\$X$ is a metavariable that represents a matched portion in the code that must be kept when the code is substituted. The formal syntax and the semantics of rewriting rules is discussed in the Chapter 4.

$$Class\ \ TreeItem:$$

$$\$X.removeAll() \equiv \$X.setItemCount(0)$$
$$\$X.removeAll() \equiv \$X.dispose()$$
$$\$X.setItemCount(0) \equiv \$X.dispose()$$

Listing 3.13. Rewriting rules

We designed an approach applicable to potentially any kind of software systems. A major challenge in any software system is to assure the state consistency in case of failure. Sometimes, failures are raised when a fault corrupts the state of the system, and the system behavior becomes unpredictable and often different from what the user expects. Moreover, the fault and the failure can be arbitrarily far from each other, and a corrupted state may persist for an indeterminate interval of time in the execution. In any case, before any action to avoid the failure can be taken, the approach must recover the state of the system to a non-corrupted instance.

JAW handles the state of the system and assures its consistency in case of failure, by saving the state of the system periodically during the execution. The state can be saved with several strategies: in predetermined points during the execution, with regular time frames, or after particular events. Each strategy has both advantages and disadvantages, and may work better under particular circumstances. Any state handling mechanism must face the problems of efficiency: saving the state too frequently makes the recovery algorithm more effective, but decreases the runtime performance of the system; few saving points have lesser impact on performance but may miss some important states that must be saved. Our state handling strategy uses the knowledge about the intrinsic redundancy of the system. For instance, in the rewriting rules in

Listing 3.13, we see that all the three methods have at least one equivalence, thus we save the state of the system before the redundant statements, where we have alternative executions if it is needed. In general, we save the state only before executing statements for which we have alternative executions. With this strategy we save the state only when it is convenient, and we minimize the code that have to be re-executed when a failure occurs and the system is recovered. The alternative strategies and techniques to handle the state are discussed in Chapter 5.

A system may fail in several different ways. Different kind of misbehaviors can be detected by different techniques. Here we assume that a failure detection technique is available and alerts us in case of failure. In general we do not put any limitation on the failure detection technique, and any kind of failure detection technique can be merged and integrated into the approach.

Figure 3.2 overviews the behaviour of JAW. During the normal execution, the state of the system is saved before executing a redundant statement. When a failure occurs and it is detected, JAW recovers the system to the most recent saved state (arrow number 1 in the figure), selects and applies a rewriting rule among the ones that apply to the redundant statement in between the recovered state and the failure point, and executes the code with the new statement. For instance, let us consider the program in Listing 3.4 that fails in removing only the children of a given item in a tree. We can assume that the state of the system has been saved, for the last time, before executing the statement *item1.removeAll()* (line 10), and that a failure detector raises an alert after the execution of that statement, at line 12. JAW recovers the state of the system at it was before the call to *removeAll()*, so the system is recovered as it was before exectuing line 10. Then, given the set of rewriting rules in Listing 3.13, JAW selects one of the two rules that apply to the portion of code included between lines 10 and 12. We call this portion of code, *frame*. The code is rewritten according to the rule and the frame is re-executed.

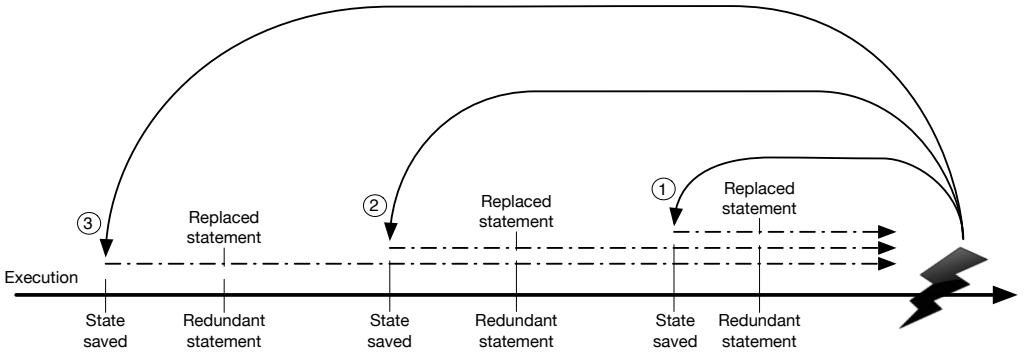


Figure 3.2. General overview of the approach JAW

If the failure detector still returns an alert signal, the approach repeats the actions represented with arrow 1 applying a new rule, until no more rewriting rules are available. If the failure still occurs, the state of the system is recovered to a previous saved point, if it exists, represented with arrow 2 in Figure 3.2. Now, the frame taken into account becomes bigger: it includes all the code between the new recovered point and the failing point. In this new frame there is at least one new rewriting rule that applies. This process continues until it either finds a workaround or exhausts the available rewriting rules. If it does not find a valid workaround, it consider a bigger frame (arrow 3) and repeats. Details and limitations of this mechanism are discussed later in Chapter 6.

The novelty of Java Automatic Workaround consists in using the intrinsic redundancy of software systems to generate workarounds automatically. Differently from other techniques, our approach does not require to either develop multiple versions of the system or part of it, or write specific code to handle failures.

In the next section we show how we applied the approach the Web applications domain.

3.3 Self-Healing for Web Applications

We applied the approach proposed in the previous section to the domain of the Web applications [CGPP10a]. In this domain we can make some simplifying assumptions. Web pages are designed to avoid side-effects caused by repeated executions of the same page, so we can assume that Web applications are stateless[3]. This way, a state recovery strategy is no longer needed. Web applications are highly interactive by nature, so we can assume that the user can identify a failure in the loaded page, and we can rely on the users as oracles: when a page fails, the user reports the failure, and the healing process starts. Finally, when a failure occurs, we can safely assume that the related fault is located in the last executed page.

We developed a prototype for JavaScript Web APIs and we carried out some experiments to demonstrate the efficacy and the efficiency of the approach applied to the Web domain. The experiments show that Web applications are, indeed, intrinsically redundant, and that we can exploit the intrinsic redundancy to automatically, even though with some limitation, efficiently avoid failures.

In the next chapters we illustrate in details the key elements of JAW: We present a qualitative and a quantitative study on the intrinsic redundancy, how we extract it and express it through rewriting rules and code rewriting rules, and their formalization in Chapter 4. We present comparison among different state handling mechanisms in showed in Chapter 5. The process to apply rewriting rules to frame of code is discussed in Chapter 6. Finally the Java prototype and the evaluation on several open source application is presented, respectively, in Chapter 7 and Chapter 8.

[3]We do not consider AJAX code.

Chapter 4

Intrinsic Redundancy

Modern software systems are naturally redundant. Modularization, re-engineering, and best development practices are only some causes of the proliferation of redundant code. This code has not been developed for failure avoidance purposes and is hidden in the program, it must be identified, extracted, and encoded. This chapter proposes qualitative and quantitative reasons to demonstrate the presence of redundancy, and shows how redundant code can be extracted, represented, and used within our approach.

This work is based on the hypothesis that failures might be avoided by means of workarounds. A workaround consists of a sequence of operations that are equivalent to a sequence of failing operations, but that do not suffer from the same failure. More specifically, a workaround consists of operations with both the same intended behavior, from the specifications viewpoint, and the same expected effect for the user's perspective, of the operations that lead to a failure. Obviously, if the two equivalent sequences of operations were *identical*, they would both fail. Thus, they have to differ to some extent. A workaround requires redundancy, that is, it requires to expose the same functionality with different code. Differently from most fault tolerance techniques, such as N-Version Programming, where the redundancy is intentionally added by designing and developing the same functionality multiple times, we argue that in modern software systems redundancy is unintentionally added by the developers for reasons that are completely extraneous to reliability purposes. We call this kind of redundancy *intrinsic*. With our technique, we show that we can take advantage of this property of modern software systems to improve the reliability of the system at a few cost.

In this chapter we show that modern software systems, especially modular systems, are indeed intrinsically redundant, and to what extent. We give some qualitative and quantitative arguments to support our intuition: we show the nature of the intrinsic redundancy, and that its presence is significant enough to be exploited for our purposes. We capture the intrinsic redundancy of software systems at the level of methods by means of *rewriting rules*, by looking at their functional behavior. Rewriting rules

abstract from the internal aspects of the methods and focus on the equivalence of the visible results, that is they relate different code fragments (method calls) that are functionally equivalent, but developed with different code.

4.1 Nature of Intrinsic Redundancy

Software redundancy has already been studied in several software engineering fields. Recent studies show that equivalent fragments of code are naturally present in software system. For example, a study on the naturalness language of programming languages shows that developers regularly write repetitive and predictable syntactically equivalent fragments of code [HBS+12]. This kind of practice may be harmful because it can lead to a proliferation of code clones. Developers often rely on bad programming practices, for instance copying and pasting code snippets to duplicate functionality across the system, deteriorating the quality of the system and making its maintenance more difficult and expensive. Notkin *et al.* show that, on average, 10% of the code in software systems is cloned [KSNM05]. Because of the deleterious consequences of code clones, many researchers propose several approaches and tools to identify and remove them, dived by the kind of technique they use to analyze the code, for example textual and lexical approaches, or syntactical and semantical approaches. Roy *et al.* classify the kind of clones and propose a comparative survey on clone detection approaches and tools [RCK09]. This kind of redundancy is harmful and error prone, and it cannot be used to improve the reliability of the system in any way, so we do not consider code clones as a good or useful form of redundancy.

But code clones are not the only form of redundancy in software systems: some studies show that there are many semantically equivalent fragments of code that are syntactically different [JS09]. In the rest of this section we give qualitative explanations on the reasons that lead to the presence redundant code in software systems.

Backward compatibility When developers release new versions of the system, they might assure backward compatibility. This is particularly true for software libraries that are used as third-party components. The developers of such libraries have to maintain different versions of the code for the same functionality to guarantee the robustness of systems that rely on those libraries after an update. For example, in the last three major versions of the Java Development Kit, 87 classes and 1065 methods have been deprecated[1]. This means that most of those classes and methods have been re-implemented, even though their functional behavior remains identical. To guide the developer towards the new implementation, the old versions of such classes and methods is kept and it is still accessible. This process duplicates the functionalities, and thus introduces redundancy.

[1] `http://www.oracle.com/technetwork/java/javase/documentation/api-jsp-136079.html`

For instance, let us consider the package *SAX*[2] from the Java 7 standard library, which provides a set of utilities to deal with XML code. The class *AttributeListImpl* belongs to the SAX package and its interface, *AttributeList*, have been deprecated and replaced by a new interface *Attributes* and its implementation class *AttributesImpl*. Both classes AttributeListImpl and AttributesImpl are still present in the code and available to the developers; they expose the same functionality, but they do not share the same code, thus they are redundant.

The widget toolkit AWT is another example. The class *List*, which shows to the user a scrolling list of text items, exposes a method called *addItem(String)* to add an item to the the scrolling list. This method has been deprecated in favor of the method *add(String)*. Again, the two methods are still present and accessible through the APIs, they provide the same functionality but are implemented with different code.

Portability and Reusability It often happens that within a library the developers implement different variants of the same functionality to increase portability and reusability. Application frameworks are a typical example. They are designed to make the development of the applications faster; the functionalities they expose have to be easily adaptable to a large variety of needs and applications. For example, in the previous work on Web applications, we studied the intrinsic redundancy in JQuery, a framework to develop dynamic and interactive Web applications. JQuery provides several routines to show an element in a Web page: the functions *fadeIn()*, *show()*, *fadeTo()*, and *animate()* are all equivalent in their final result, that is to make an element to appear in the page, and all of them can be easily exchanged.

Widget toolkits, and collections frameworks are also typical and interesting examples. Widget toolkits usually provide functionalities to draw shapes, such as rectangles and lines. For instance, the SWT library exposes two dedicated methods for this purpose, *drawRectangle(Rectangle rect)* and *drawLine(int x1, int y1, int x2, int y2)*. There exists also another method, *drawPolygon(int[] pointArray)*, which depending on the numbers of points given in the array as parameters, can draw the corresponding shape. Other representative cases in SWT are the method *fillRectangle(Rectangle rect)* that fills the interior of the specified rectangle, and the equivalent method *fillPolygon(int[] pointArray)*, or the methods *drawText(String string, int x, int y)* and *drawString(String string, int x, int y)*that draw a given string. Intuitively, it is possible to match pairs of methods that can be easily exchanged.

The same modus operandi is applicable to collections libraries. These libraries provide sets of utilities to handle with collections such as List, Map, and HashMap. For instance, let us consider the Guava project[3], a container of several Google's core libraries including collections, caching, and I/O. The collection package of Guava provides a functionality to check whether a collection is empty or not, the method *isEmpty()*.

[2] `http://docs.oracle.com/javase/7/docs/api/org/xml/sax/helpers/package-summary.html`
[3] `https://code.google.com/p/guava-libraries`

Each collection also exposes the method *size()*, which intuitively can be used to check the same property. Once more, functionalities such as removing all the elements of a given collection, can be achieved by using specialized methods, such as *removeAll()*, or by removing each element singularly with a *remove()* method.

These design and development practices are a large source of redundant code.

Third-Party Components Similarity Reusability influences not only the way components are designed and developed, but also the way they are used. Many popular utilities or set of utilities are often implemented by several libraries. The large variety of these libraries offer unique features, such as complex mathematical analysis or specific graph visualization, but they share many standard features to enable their distinct functionalities, for example simple mathematical operations or primitives to model or analyze graphs or networks. Given each library unique characteristic, often developers include more than one library to access the different features. Given the common set of functionalities that the libraries share, those systems include many duplicated functionalities. Such duplicated functionalities are often easily exchangeable even across different libraries. For instance, let us take the function to compute the absolute value of a number, *Math.abs(double a)*, of the standard Java mathematical package, *java.lang.Math* that "contains methods for performing basic numeric operations such as the elementary exponential, logarithm, square root, and trigonometric functions"[4]. The functionally equivalent operations *abs* is duplicated in a large set of alternative mathematical libraries for Java: Apache Commons Math, "a library of lightweight, self-contained mathematics and statistics components"[5], exposes the method *Abs.value(double x)*; Apfloat, "a high performance arbitrary precision arithmetic library"[6], provides *ApfloatMath.abs(Apfloat x)*; Colt, "a set of Open Source Libraries for High Performance Scientific and Technical Computing"[7], exposes *Functions.abs()*; Java Ultimate Math Package, "a framework for arbitrary precision computations"[8], provides *IntegerNumber.abs()*; JMSL, "a collection of mathematical, statistical and charting classes"[9], has a *Jampack.abs(Z z)* method; finally, JSci," a set of Java packages for linear algebra and statistics"[10], exposes *ArrayMath.abs(double[] v)*.

A similar situation occurs with graph analyzer components. There are many Java libraries that model, analyze, and visualize graphs and networks, including JGraphT, JUNG, yWorks, and JGraph. Such libraries are usually specialized in some particular features, but they offer large set of common functionalities, such as creating a graph from a matrix or vice-versa.

[4] http://docs.oracle.com/javase/7/docs/api/java/lang/Math.html
[5] http://commons.apache.org/math
[6] http://www.apfloat.org
[7] http://acs.lbl.gov/software/colt/
[8] http://jump-math.sourceforge.net
[9] http://www.aertia.com/en/productos.asp?pid=238
[10] http://jsci.sourceforge.net

Non-Functional Requirements Non-functional requirements are, in some domains, a crucial factor. Efficiency, performance, or scalability problems might suggest to the developers that some part of the system should be re-engineered, even though the existing functionality works as expected in terms of expected functional results. For example the sorting algorithm in the standard C++ library is implemented, among several, in two main variants, namely, *std::sort* and *std::qsort*. The complexity of both the algorithms is $O(N \cdot log(N))$, but Meyers, in his book, claims that the std::sort algorithm is about 670% faster than the std::qsort algorithm, because it uses *inline* calls to the sorting comparison function, which are known to be faster than normal calls to functions [Mey01].

The Apache Commons Math library offers another example. The library exposes three methods that implement three different algorithms to compute the roots of an equation: *Regula Falsi*, *Illinois*, and *Pegasus* methods. The three algorithms are functionally equivalent in finding the root of an equation, but according to the documentation "the Illinois method however, should converge much faster than the original Regula Falsi" and "the Pegasus method should converge faster than the Illinois method".

Yet another example can be found in the Ant library, a Java tools and library to simplify the building process of big projects. The library exposes a method to break up a path String into a Vector of path elements delimited by a separator token, *tokenizePath(String path)*. Interesting, the same class exposes another method, called *tokenizePathAsArray(String path)* that, as the documentation clearly explains, works the "same as tokenizePath but faster".

Often, developers develop specialized libraries that duplicate an entire set of functionalities, already provided by some standard or third-party library, to achieve better performance. For instance, the Java standard package *java.util.Collections* can be easily replaced with, at least, three other libraries that duplicate every functionality of the standard library, but they perform better, Guava, Trove and Colt. Another common case of a library that perform better than a standard feature of a language is *Log4J*, a Java logging library. Log4J is widely used in place of the standard *java.util.Logging* due to the increasing of the performance. Yet another example, the same developers of Log4J also developed another logging library for Java, *logback*, which performs better than Log4J.

So far we presented some qualitative reasons that justify the presence of intrinsic redundancy in modern software systems, especially in libraries and reusable components. In the rest of the chapter we present some examples of intrinsic redundancy that we can find in modern software systems. Then we introduce rewriting rules to code such redundancy, their implementation for Java and a formal definition of their syntax and semantics. We will conclude this chapter with a study on the intrinsic redundancy on some large Java libraries, with quantitative data to support our intuition.

4.2 Redundancy in Source Code

In the previous section, we introduce some qualitative reasons that lead to redundant systems. Redundant functionalities produce the same results, as described in the specifications, and bring the system to the same state so that, independently from the *version* of the functionality used to achieve the result, the user of the system will obtain the expect behavior.

The degree of the redundancy in source code may vary and it can affect the effectiveness of JAW. Informally, starting from the lower degree of redundancy, we can observe the *code identity*. In this case, the portion of code is exactly the same and its re-execution will provide the same results, if the system is deterministic. We can then observe the *code equality*, where the portion of code is replicated or cloned, for instance as a consequence of a copy and paste. In this case we have distinguishable pieces of code, but that represent exactly the same code and thus we expect that they behave the same. These two instances are not interesting in our work because, since the executed code is exactly the same, they do not provide a way to avoid a failure as discussed in Chapter 3.

We observe that the redundancy can also originate from portions of different code that produce the same results. This is the type of redundancy we are interested in. Such portions of code bring the system to a state that is indistinguishable from the user viewpoint, which means that from that point on, the system will behave exactly the same no matter how the point was reached. The redundant portions of code may be entirely or partially different. The more dissimilar they are, the more likely they can provide an effective way to avoid a failure.

This intrinsic redundancy can be identified from the component interfaces. The documentation of an interface usually provides the information to understand if two or more methods have the same behavior. This information is essential to enable the code substitution and preserve the original behavior of the system. What this information does not usually say is how much the two or more methods differ in terms of source code. The rest of this section illustrates two examples of redundant methods, one barely and one fully redundant. The examples focus on intra-component redundancy, that is the redundancy present within a component.

The Java library JodaTime offers an example of a poorly redundant code. JodaTime is a replacement for the Java *date* and *time* classes. It provides features to handle with different calendars, timezones, and to operate with dates and time, such as time period calculation or conversion. The class *AbstractConverter* simplifies the creation of a custom converter for time. A time type, intuitively, is specified by fields that characterize the date, such as the hours, the minutes, the seconds, and the timezone. The AbstractConverter interface exposes, among the others, two methods to extract the values of the fields of partial times (a time that does not support every time field, such as the timezone, and is thus a local time). The two methods are *get-*

PartialValues(ReadablePartial fieldSource, Object object, Chronology chrono) (from now on, getPartialValues′) and *getPartialValues(ReadablePartial fieldSource, Object object, Chronology chrono, DateTimeFormatter parser)* (from now on, getPartialValues″), respectively with three and four parameters. They differ for the last parameter, and the documentation of the methods says that both "extract the values of the partial time from an object of this converter's type", thus given a standard DateTimeFormatter as parameter, they are functionally equivalent and can be used interchangeably.

From the implementation of these methods, reported in Listing 4.1 and 4.2, we can see that the method getPartialValues″ (Listing 4.2) simply calls its overloaded method getPartialValues′ (Listing 4.1), thus, the executed code is fundamentally the same. If a failure occurs in getPartialValues′, it is very likely that getPartialValues″ suffers from the same failure.

```
1 public int[] getPartialValues(ReadablePartial fieldSource, Object object, ↩
      Chronology chrono) {
2  long instant = getInstantMillis(object, chrono);
3  return chrono.get(fieldSource, instant);
4 }
```

Listing 4.1. One implementation for the method *getPartialValues*

```
1 public int[] getPartialValues(ReadablePartial fieldSource, Object object, ↩
     Chronology chrono, DateTimeFormatter parser) {
2  return getPartialValues(fieldSource, object, chrono);
3 }
```

Listing 4.2. Another implementation for the method *getPartialValues*

To be effective in avoiding failures by means of intrinsic redundancy, the methods have to provide the same functionality and expected behavior, but they also have to execute different code. For instance, let us consider the class *LinkedListMultimap* provided by the Guava library. A LinkedListMultimap is a collection that maps keys to multiple values. The LinkedListMultimap class exposes the method *containsKey(Object key)* that returns true if the given LinkedListMultimap contains a values for the specified key. Inspecting the interface of the LinkedListMultimap, we find two interesting methods: *keys()*, which returns the collection of all the keys of the LinkedListMultimap; *keySet()*, which returns the set of all keys of the LinkedListMultimap. Then the Collection and Set classes expose the methods *contains(Object key)* and *containsAll(new Collection<T>(key))* to determine whether they contain the specified elements. Coupling these last four methods correctly, we can derive four alternative ways to verify the existence of an element in a LinkedListMultimap:

keys().contains(key)

keys().containsAll(new Collection<T>(key))
keySet().contains(key)
keySet().containsAll(new Collection<T>(key))

Let us focus on the original call *containsKey* and the first alternative sequence, the sequence of calls *keys* and *contains*, to understand the level of redundancy and the differences in the executed code. Listing 4.3 is the implementation of the method containKey in the class LinkedListMultimap.

```
1 import java.util.Map;
2 private transient Map<K, Node<K, V>> keyToKeyHead;
3
4 public class LinkedListMultimap<K, V> implements ListMultimap<K, V>, ↩
      Serializable {
5   @Override
6   public boolean containsKey(@Nullable Object key) {
7     return keyToKeyHead.containsKey(key);
8   }
9 }
```

Listing 4.3. Implementation of method *containKey* of class *LinkedListMultimap*

If we compare the implementation of method *containKey* with the implementation of methods *keys* (Listing 4.4) and then *contains* (Listing 4.5), we notice that they are completely different. In this case, a failure in method *containKey* might be avoided by executing the methods *keys* and then *contains* that execute different code.

```
1  public class LinkedListMultimap<K, V> implements ListMultimap<K, V>, ↩
       Serializable {
2   @Override
3   public Multiset<K> keys() {
4     Multiset<K> result = keys;
5     if (result == null) {
6       keys = result = new MultisetView();
7     }
8   return result;
9   }
10 }
```

Listing 4.4. Implementation of method *keys* of class *LinkedListMultimap*

4.3 Rewriting Rules

We encode the intrinsic redundancy at the level of method calls by means of rewriting rules. A rewriting rule substitutes a code fragment with a different redundant one,

```
1  abstract class AbstractMultiset<E> extends AbstractCollection<E> implements ↩
       Multiset<E> {
2    @Override
3    public boolean contains(@Nullable Object element) {
4      return count(element) > 0;
5    }
6
7    @Override
8    public int count(Object element) {
9      for (Entry<E> entry : entrySet()) {
10       if (Objects.equal(entry.getElement(), element)) {
11         return entry.getCount();
12       }
13     }
14     return 0;
15   }
16 }
```

Listing 4.5. Implementation of method *contains* of class *AbstractMultiset*

that is a fragment with the same observable behavior of the original one. We thus abstract away from internal aspects, such as details of the internal state that do not affect the observable behavior of the system, and non-functional properties, such as performance or usability. A rewriting rule encodes the fragment of code to be matched in the system, the fragment of code that substitutes, and it indicates how to rewrite the first fragment to produce a new one, which is equivalent in the intended effects, to the original one. Rewriting rules do not depend on any programming language, but they must rewrite the code fragment considering actual variables and parameters used in the original code fragments.

A rewriting rule is composed of a context and two patterns: the first pattern matches the original code fragment, the second pattern contains the new code fragment that substitutes the original one, the context restricts the applicability of the rule. The patterns are characterized by a constant part and some meta-variables. The constant part matches the pattern of the rewriting rule in the original source code, the meta-variables stores and preserves the variable parts in the original code fragments that must be rewritten in the new code fragment.

In the previous section we introduced two equivalences: one from the Java library JodaTime, the method *getPartialValues* from the class *AbstractConverter*, and one from the Java library Guava, the method *containsKey* in the class *LinkedListMultimap*. We express the two equivalences as rewriting rules in Listing 4.6 and 4.7, respectively.

The rewriting rule in Listing 4.6 captures the equivalence between the two methods *getPartialValues* of the JodaTime library in the context of the *AbstractConverter* class. The rule applies to every code fragment of a source code that refers to the method

$$
\begin{aligned}
&Class\ AbstractConverter: \\
&\$X.getPartialValues(\$P_1,\ \$P_2,\ \$P_3) \equiv \\
&\$X.getPartialValues(\$P_1,\ \$P_2,\ \$P_3,\ newDateTimeFormatter)
\end{aligned}
$$

Listing 4.6. Rewriting rule for the JodaTime getPartialValues method

getPartialValues. The meta-variables try to match the rest of the code by following the rule pattern, that is some code $\$X$ before the name of the method and some code $\$P_1$, $\$P_2$, and $\$P_3$ within brackets and separated by commas.

$$
\begin{aligned}
&Class\ LinkedHashMap: \\
&\$X.containsKey(\$P_1) \equiv \$X.keys().contains(\$P_1) \\
&\$X.containsKey(\$P_1) \equiv \$X.keys().containsAll(newCollection(\$P_1)) \\
&\$X.containsKey(\$P_1) \equiv \$X.keySet().contains(\$P_1) \\
&\$X.containsKey(\$P_1) \equiv \$X.keySet().containsAll(newCollection(\$P_1))
\end{aligned}
$$

Listing 4.7. Rewriting rules for the Guava containsKey method

Listing 4.7 presents the four rewriting rules that we can derive from the description of the equivalent codes given in the previous section for the method *containsKey* in the *LinkedListMultimap* class of the Guava library.

To show how the rewriting rules apply to the source code, let us consider Listing 4.8, a method that splits a string into substrings using a delimiter, and each substring into an entry for a map. If we apply the first of the four rewriting rules in Listing 4.7, the code will be affected as follow: the constant part of the rule, *containsKey*, matches line 7, that is $checkArgument(!map.containsKey(key),\ key)$, and two metavariables match as follows: $\$X := map$ and $\$P_1 := key$. Thus the code can be rewritten as $checkArgument(!map.keys().contains(key),\ key)$. All the four rewriting rules in Listing 4.7 can be applied to the code in example, Listing 4.9 shows the results.

4.4 Code Rewriting Rules for Java

Rewriting rules are a general and language independent representation of the intrinsic redundancy. To exploit this knowledge within a real system, we need to encode the rewriting rules into a language that is compatible with a programming language. We call such language dependent rewriting rules, *code rewriting rules*.

```
1  public Map<String, String> split(CharSequence sequence) {
2    Map<String, String> map = new LinkedListMultimap<String, String>();
3    for (String entry : outerSplitter.split(sequence)) {
4      Iterator<String> entryFields = entrySplitter.spliterator(entry);
5      checkArgument(entryFields.hasNext(), INVALID_ENTRY_MESSAGE, entry);
6      String key = entryFields.next();
7      checkArgument(!map.containsKey(key), key);
8      checkArgument(entryFields.hasNext(), INVALID_ENTRY_MESSAGE, entry);
9      String value = entryFields.next();
10     map.put(key, value);
11     checkArgument(!entryFields.hasNext(), INVALID_ENTRY_MESSAGE, entry);
12   }
13   return Collections.unmodifiableMap(map);
14 }
```

Listing 4.8. A code fragment that contains the *containsKey* method

$$checkArgument(!map.containsKey(key),\ key);$$
$$\downarrow$$
$$checkArgument(!map.keys().contains(key),\ key);$$
$$or$$
$$checkArgument(!map.keys().containsAll(new\ Collection(key)),\ key);$$
$$or$$
$$checkArgument(!map.keySet().contains(key),\ key);$$
$$or$$
$$checkArgument(!map.keySet().containsAll(new\ Collection(key)),\ key);$$

Listing 4.9. The method *containsKey* rewritten by means of the four rewriting rules

Code rewriting rules are specialized for a specific language. In this work we focus on Java applications, so our code rewriting rules work with Java source code. Code rewriting rules are regular expressions: the constant parts of the rewriting rules remain the same, the meta-variables are encoded as capturing groups, usually written as regular expression character classes. The developer can write their own character classes, or use the Java predefined classes, such as \w for word characters, or \S for non-whitespace characters. For instance, we can derive a code rewriting rule for the rewriting rule of method *getPartialValues* of JodaTime, in Listing 4.6, by transforming the meta-variables $\$X$ and $\$P_{1..3}$ into regular expression capturing groups (Listing 4.10). The expressiveness of the code rewriting rules is bounded to the expressiveness of the regular expressions: a code rewriting rule will match a code fragment depending on the richness of the character classes implementing the meta-variables of the rewriting rule.

$$
\begin{aligned}
&Class\ AbstractConverter: \\
&\$X.getPartialValues(\$P_1,\ \$P_2,\ \$P_3) \equiv \\
&\$X.getPartialValues(\$P_1,\ \$P_2,\ \$P_3,\ newDateTimeFormatter) \\
&\Downarrow \\
&([\backslash w\$.()"\ ]*)\backslash .getPartialValues\backslash (([\backslash S]*),\ ([\backslash S]*),\ ([\backslash S]*)) \equiv \\
&\$1.getPartialValues(\$2,\ \$3,\ \$4,\ newDateTimeFormatter())
\end{aligned}
$$

Listing 4.10. Code rewriting rule for the JodaTime *getPartialValues* method

Code rewriting rules might be more general than rewriting rules. Let us take two Guava classes: *ForwardingMultiSet* and *ForwardingList*. These two methods expose a method called *removeAll*. The method exposed by the former class removes all the elements in the set, while the method exposed by the latter removes all the elements in the list. Both classes expose also two methods called *standardRemoveAll*, which is equivalent to the previous methods for both classes, respectively. We can express this knowledge with the two rewriting rules in Listing 4.11.

We can encode the two rewriting rules in Listing 4.11 into one code rewriting rule that is more general than each singular rewriting rule, but that expresses the same knowledge and includes both of them (Listing 4.12).

Regular expressions have limitations: we cannot substitute code fragments based on their dynamic context, because the dynamic information cannot be included into the code rewriting rules, or it might be difficult to match long multi lines pattern that are interleaved with other lines of code. But regular expressions can generally be written

$$Class\ ForwardingMultiSet:$$
$$\$X.removeAll(\$P_1) \equiv \$X.standardRemoveAll(\$P_1)$$

$$Class\ ForwardingList:$$
$$\$X.removeAll(\$P_1) \equiv \$X.standardRemoveAll(\$P_1)$$

Listing 4.11. Rewriting rules for the method *removeAll* in two different contexts

$$([\backslash w\$.()"\ ]*)\ \backslash\ .removeAll\ \backslash\ (([\backslash S]*)\backslash) \equiv \$1.standardRemoveAll(\$2)$$

Listing 4.12. Code rewriting rule that implements the two rewriting rules in Figure 4.11 for *removeAll*

with a small effort and are flexible enough to allow a wide set of code matching. Chapter 7 explains in details how we deal with techniques such as polymorphism and subtyping.

In the code rewriting rules we wrote in this work, we assume that the source code of the Java systems has been formatted following the style guidelines provided by Oracle[11]: for example a space divides a comma to the next parameter; or the conditional and loop statements are separated to their condition within the parentheses by a space; or a return-carriage comes after every bracket. This assumption can be overcame by normalizing the source code of the target system to match the standard guidelines so that the coding style of the developers will not affect the efficacy of the code rewriting rules.

4.5 Syntax and Semantics of Code Rewriting Rules for Java

In this section we formalize the code rewriting rules for Java by means of Featherweight Java [IPW01]. We formalize Java, the patterns that occur in the code rewriting rules and the semantics of the code rewriting rules.

Java We first define a grammar for the abstract syntax of the essential elements of Java borrowing them from Igarashi *et al.* [IPW01].

Grammar 4.1 formalizes some elements of the Java language. The variable L ranges over class declarations; K ranges over constructor declarations; and M ranges over method declarations. The variable C spans over class names; f spans over field

[11] `http://www.oracle.com/technetwork/java/codeconv-138413.html`

$$
\begin{aligned}
L &::= \text{class } C \text{ extends } C\ \{\bar{C}\ \bar{f};\ K\ \bar{M}\} \\
K &::= C(\bar{C}\ \bar{f})\{\text{super}(\bar{f});\ this.\bar{f} = \bar{f};\} \\
M &::= C\ m(\bar{C}\ \bar{x})\{\text{return } e;\} \\
e &::= x \mid s \mid let\ x = e\ in\ e \mid e.f \mid e.f = e \mid e.m(\bar{e}) \mid \text{new } C(\bar{e})
\end{aligned}
$$

Grammar 4.1. Grammar for some Java elements

names; m spans over method names; x spans over variables (including $this$); e represents expressions; s represents strings. We also borrow from Featherweight Java the notation $\bar{f}$ as shorthand for a possibly empty sequence $f_1, \ldots, f_n$ (and similarly for $\bar{C}$, $\bar{x}$, $\bar{e}$, etc.) and write $\bar{M}$ as shorthand for $M_1 \ldots M_n$ (with no commas).

We take the definition of L, K and M from Featherweight Java, while we slightly modify the definition of the expressions e: an expression e can be a local variable x, a let expression that introduces new local variables x in an expression e, a string literal s, a field accesses $e.f$, a field assignments $e.f = e$, a method call with zero or more parameters $e.m(\bar{e})$ or an instantiation of new objects through a call to a constructor with zero or more parameters $new\ C(\bar{e})$.

We then define a *rewriting context* E for Grammar 4.1 inductively. The rewriting context is an "expression with a hole". The hole is filled with an expression to obtain another expression, that is, the hole represents an element to be rewritten, and therefore is the basis for the code rewriting rules.

$$
\begin{aligned}
E ::= &[\bullet] \mid let\ x = E\ in\ e \mid let\ x = e\ in\ E \mid E.f \mid x.E \mid x.E(\bar{e}) \mid \\
&e.f = E \mid E.m(\bar{e}) \mid e.m(\bar{e}, E, \bar{e}) \mid \text{new } C(\bar{e}, E, \bar{e})
\end{aligned}
$$

Grammar 4.2. The rewriting context

Grammar 4.2 defines the rewriting context E. For each expression there can be only one hole $[\bullet]$ that represents an element to be rewritten. The hole in the expression can be located in an instantiation of a new local variable ($let\ x = E\ in\ e$), an object that accesses a field ($E.f$), a field of an object ($x.E$), a method called on an object ($x.E(\bar{e})$), a value stored in a field ($e.f = E$), an object on which a method is called ($E.m(\bar{e})$), a parameter of a method ($e.m(\bar{e}, E, \bar{e})$), or a parameter of a constructor ($new\ C(\bar{e}, E, \bar{e})$).

We will use rewriting contexts to define which parts of a program may be changed by a code rewriting rule. Let us consider for example a source program fragment $P = object.methodA()$ that calls a method $methodA()$ of an object instance variable $object$, and a target program $P' = object.methodB()$. The transformation between

P into P' effectively replaces $methodA()$ with $methodB()$. This replacement corresponds to the rewriting context $object.[\bullet]$, which is consistent with Grammar 4.2 and therefore acceptable.

Pattern The core of our code rewriting rules are pairs of patterns $\langle pattern, pattern' \rangle$, where a $pattern$ is an expression that matches a code fragment of a Java application and $pattern'$ is the code fragment that substitutes $pattern$ when applying the rule. In the code rewriting rules, in addition to normal Java expressions, $pattern$ can contain meta-variables taken from a given set $mv(p) = \{\$X_1, \ldots, \$X_k\}$. Thus a pattern is a subset of Java expressions augmented with meta-variables.

$$p ::= x \mid p.m(\bar{p}) \mid s \mid new\ C(\bar{p}) \mid \$X$$

Grammar 4.3. Grammar for code rewriting rules patterns with meta-variables

A pattern p (defined in Grammar 4.3) can be a local variable x, a method call with zero or more parameters $p.m(\bar{p})$, a string literal s, a parameter in a constructor ($new\ C(\bar{p})$) or a meta-variable $\$X$.

Code Rewriting Rule We now define a code rewriting rule in Grammar 4.4. The keywords ANY and ALL define the scope of the rule within the program code. If the scope is ANY, the transformation is applicable to any one of the occurrences of the substitution pattern and therefore it can produce several new programs, each differing from the original in one specific application of the transformation. If the scope is ALL, the rule applies to all the occurrences of the substitution pattern, and therefore may produce only one new program.

$$r ::= ANY\ (p \mapsto p') \mid ALL\ (p \mapsto p')^{+}$$

Grammar 4.4. Grammar for code rewriting rules

The mapping $(p \mapsto p')$ defines a transformation from a pattern p, called substitution pattern, to a pattern p', called replacement pattern. The meta-variable in the replacement pattern refers to the corresponding meta-variables in the substitution pattern, thus we require for all rules $p \mapsto p'$ that $mv(p') \subseteq mv(p)$.

Application of Code Rewriting Rules In order to define the semantics of a code rewriting rule, consider a general ANY rule $r = ANY\ (p \mapsto p')$ of scope ANY, and transformation patterns $(p \mapsto p')$, and without loss of generality let $mv(p) = mv(p') =$

$\{\$X_1, \ldots, \$X_k\}$. Given a program P, rule r applies to any fragment $\tilde{P}$ within P if, for each meta-variable X_i in p there exists a rewriting context E_i in $\tilde{P}$ such that $\$X_i$ matches the hole in E_i, and the rest of $\tilde{P}$ matches the rest of pattern p completely, token by token. We indicate this condition through Listing 4.13 where $p[\$X_1 = e_1, \ldots, \$X_n = e_n]$ indicates pattern p in which a concrete expression e_i replaces each meta-variable $\$X_i$.

$$P = P_L \cdot \tilde{P} \cdot P_R \quad \text{where} \quad \tilde{P} = p[\$X_1 = e_1, \ldots, \$X_n = e_n]$$

Listing 4.13. Matching condition for code rewriting rules

Listing 4.14 represents, then, the rewritten program code, that is, program fragment $\tilde{P}$ in P is rewritten as $\tilde{P}'$ corresponding to the replacement pattern p' in which each meta-variables $\$X_i$ is replaced with the concrete code e_i bound to $\$X_i$ in the substitution pattern.

$$P' = P_L \cdot \tilde{P}' \cdot P_R \quad \text{where} \quad \tilde{P}' = p'[\$X_1 = e_1, \ldots, \$X_n = e_n]$$

Listing 4.14. The rewritten program code

The semantics of a code rewriting rule with scope ANY extends naturally to rules with scope ALL. Intuitively, the substitution applies to all disjoint program fragments $\tilde{P}$ where it would be applicable in an identical ANY rule. Operationally, the transformation of an ALL rule can be obtained by applying the corresponding ANY transformation to the *leftmost* applicable fragment $\tilde{P}$, and then recursively to the rest of program to the right of $\tilde{P}$. A bit more formally, let r_{ANY} and r_{ALL} be two identical rules with scope ANY and ALL, respectively, and let $r(P = P_L \cdot \tilde{P} \cdot P_R) = P_L \cdot \tilde{P}' \cdot P_R$ indicate the application of the r transformation to the leftmost fragment $\tilde{P}$ in P, then the ALL transformation can be defined recursively as $r_{\text{ALL}}(P = P_L \cdot \tilde{P} \cdot P_R) := P_L \cdot \tilde{P}' \cdot r_{\text{ALL}}(P_R)$.

4.6 Study on Intrinsic Redundancy

We methodically studied the intrinsic redundancy in four large representative Java libraries:

- *Guava:*[12] the Google "core" library that implements a framework to work with collections, I/O, caching, concurrency, string processing.
- *SWT:*[13] the open source widget toolkit for Java designed to provide a portable access to the user-interface facilities of the operating systems.

[12] `http://code.google.com/p/guava-libraries/`
[13] `http://www.eclipse.org/swt/`

- *JodaTime:*[14] a library of utility functions to represent and manipulate dates and time, designed to replace the standard Java date and time classes.
- *Lucene:*[15] it is a search engine library, which features include fast indexing, ranked searching, and span queries, date-range searching. Lucene's API can index and extract information from several file formats such as PDF, HTML, XML.

We manually inspected the interfaces of the four libraries by examining the documentation of each public method considered. We did not look at the implementation, but we considered the intended behavior documented by the developers of the libraries. We identified the equivalences within the libraries by reading the documentation of the methods of the interface and then testing their behavioral equivalence experimentally. In this study we did not applied a more insightful analysis to understand the degree of redundancy of the equivalences we found, that is we do not know how the interfaces are implemented and if two methods that we consider equivalent also execute different code. One equivalence corresponds to a knowledge that we can write as a rewriting rule.

Library	Guava	JodaTime	SWT	Lucene
Classes considered	116	12	252	563
Total equivalences found	1715	135	1494	686
Average per class	14.78	11.25	5.93	1.21

Table 4.1. Equivalent sequences found in representative Java libraries

Table 4.1 shows the results of our analysis. The table reports the number of the classes we analyzed for each library (all the SWT classes and a selected set for Guava, JodaTime and Lucene). Based on the classes we considered, the table reports the number of the equivalences we found, as a total number and as average per class.

The analysis is not exhaustive because it considers only a subset of all the classes for most of the libraries, although it is minimal in the sense the we did not consider the identity function and the biunivocal property of the rules counts only once. This means that we consider the rules $e1 = e2$ and $id(e1) = id(e2)$ as a single occurrence. In the same way, we consider as a single occurrence the rules $e1 = e2$ and $e2 = e1$.

[14] `http://www.joda.org/joda-time/`
[15] `http://lucene.apache.org`

Chapter 5

State Consistency Mechanisms

In this chapter we discuss two approaches to assure state consistency, software transactional memory and checkpoint and recovery. They have been designed and implemented to address specific problems, such as concurrency or restoration of the state of a system after an interruption. Our approach works with stateful systems, so we need a mechanism to assure the consistency of the state and to recover it after a failure. Here, we evaluate several techniques based on the two approaches to determine the most suitable one to include in our approach. To describe the techniques, we mainly focus on how they handle with the data.

In this work we consider general purpose systems. These systems are generally stateful, that is their behavior depends on the inputs and the internal state, and every action potentially changes the state of the system. When a failure happens in such systems, it might corrupt the state of the system. A corrupted state leads the system to behave differently from the specifications and the user expectation, so that every further action, or even attempts to avoid the failure are pointless or even harmful. To avoid a failure, JAW tries to substitute a failing operation with a different and potentially error-free one. Let us then assume that an operation modifies the state of the system correctly, for instance it subtracts the proper amount of money from a bank account, but it returns a failing code. JAW tries again to achieve the expected result but with a different operation, so if the bank account is not restored to its original balance, the same amount of money is subtracted again. If this procedure applies several times, we repetitively add error on error. Then, it is important to handle the state of the system, to recover it in case of failure and to assure its consistency while we try to avoid the failure.

We considered two strategies to deal with the state of the system: software transactional memory (STM) and checkpoint and recovery mechanisms. Software transactional memory has been proposed by Shavit *et al.* in the nineties [ST95], and it is mainly used in concurrent and shared-memory systems to support the developers and

provide them primitives to synchronize processes and avoid typical domain problems such as deadlock and race conditions. Transactions in software transactional memory borrow the same principles standardized in the database domain as the ACID properties [Gra81, GR92], especially atomicity and isolation. The atomicity property assures that a piece of code either runs to completion, or all its effects are discarded and are not visible to the system. The isolation property assures that the data involved in a transaction are not touched by other transactions. Changes in the state of the system are usually made visible when a transaction performs a *commit* action, or are completely discarded upon an *abort* or a *retry*. These capabilities make software transactional memory attractive for building and integrating recovery mechanisms to support self-healing approaches, but even if software transactional memory claims to able to simplify the design and the production of such appliances, the complexity it introduces and the performance issues observed limit its application [Her09, CBM+08, BDK+08].

Checkpoint and recovery is a common technique for saving and restoring the state of a system [CR72]. It has been used in traditional fault tolerance methods to recover after a failure interrupts the system, causing a task to fail or abort [Lyu95]. It is used primarily to save the whole state of the system to non-volatile storage so that, upon a failure, the state of the system can be restored and the service can start again at the point at which it was saved. This approach has been applied in a wide range of areas such as distributed environments [KT86, JZ88], fault-tolerance [RS95, WHV+95], debugging [SKAZ04, XRTQ07] and persistence and migration of virtual machines [How98, Sue00]. The main pitfalls of this approach are related to the time and the memory space required to save the state of the system along the execution. To address these problems, researchers have proposed many approaches and techniques that increase the performance, and limit the required memory space by following two main strategies: optimizing the frequency of the checkpoints [CR72, You74, Gel76] and observing which parts of the state of the system is critical to assure state consistency [PCL+99, LM00, Fen05a].

The next sections of this chapter survey several software transactional memory and checkpoint and recovery techniques, and describe the approach we decided to use to handle with the state of the systems highlighting the motivations of the choice.

5.1 Software Transactional Memory

In the early nineties, Herlihy and Moss [Her91, HM93] introduced transactional memory (later referred as hardware transactional memory) as an alternative and simpler way to support the synchronization of shared data structures, providing a new level of abstraction for concurrent programs to free the developers from the complex mechanisms of locks. Locks have always been difficult to use and error prone, often leading developers to deal with deadlocks and race conditions instead of just concentrating on the algorithm [HCU+07]. Transactional memory supports the implicit definition of

code regions that have to be considered as a transaction. A transaction is a sequence of instructions, including the access to shared-memory, that must execute atomically and in isolation. It can either execute completely, thus it commits, or aborts, having no effects on the system. All the writes that a transaction does on the memory are visible to the system only after a commit. When a transaction aborts, all the writes are discarded.

The implementation of the hardware transactional memory by Herlihy and Moss is based on a modification of the standard multiprocessor cache coherence protocols to enable transaction conflicts detection. They add a transactional cache for each processor for transactional operations. Data written in that cache are propagated to the main memory only after a commit [HM93].

While Herlihy and Moss defined the idea of transactions based on hardware support, Shavit and Touitou introduced software transactional memory. Software transactional memory is more portable and applicable to different machine and more resilient on hardware failures than the approach proposed by Herlihy. The downside is that software transactional memory pays in performance [ST95].

To support transaction execution, software transactional memory systems need a data versioning mechanism to store all the changes in the data made within a transaction and before a commit or abort operation. The two most used approaches are either a *change-log* or a *shadow copy* mechanisms. With the change-log approach, the transaction applies the changes directly to locked locations in memory, while a log of all the changes is stored separately, so that in case of an abort all the changes can be discarded (undo). On the other side, in the shadow copy approach we have two specular strategies: all the transactions, before any writes, make a private backup of the memory locations, so that in case of abort, the backup can be restored to avoid all the changes; or the transactions have a private buffer to store all the changes, and only in case of commit, such changes are propagated to the main memory and visible to the other transactions.

Adl-Tabatabai *et al.* [ATLM$^+$06] and Hindman *et al.* [HG06] propose two approaches based on *change-log*. The Adl-Tabatabai *et al.*'s approach tracks all the transactional memory accesses maintaining for each transaction a read set, a write set, and an undo log. Reads and writes are managed with two different strategies. For reads they use a versioning optimistic concurrency control, when a transaction reads a location memory, it also reads its version number; if on a following read or write operation the version number has changed, the transaction is aborted [KR81]. For writes, they apply a strict two-phase locking strategy, a transaction can only acquire locks (first phase) and then only release locks after the transaction is completed, with a commit or an abort (second phase) [GR92]. When a transaction updates memory locations, the new value is written in place, and the original value is logged to rollback side effects on an abort. To integrate software transactional memory into Java, the authors propose to add few statements into the language: some traditional transactional statements such

as *atomic*, *retry* and a specific statement, *orelse*, to provide alternatives for aborting transactions.

Similarly to the former work, Hindman *et al.* propose a change-log approach, but based on different locking and logging strategies. In this approach, each object can lock itself in the sense that every object has a field that associates the object itself to the thread that is currently owning the lock and a thread keeps the locks on the objects until the end of the transaction. To prevent deadlocks, each thread that tries to access a locked object, can ask the current holding thread to release it. In this case, if the current holder thread is blocked, or no longer alive, its resources can be safely released with a proper synchronization. The approach logs all the write operations in a conceptual stack so that, to rollback in case of abort, it pops the elements off of the stack and re-assigns the old values to the objects. The authors optimize the logging operations by avoiding duplicates: when a transaction updates the same memory location multiple times, only the first needs a log entry.

Herlihy *et al.* propose a software transactional memory technique based on the *shadow copy* approach. The approach implements a particular lock-free synchronization technique called obstruction-free. This strategy guarantees that a single thread executed in isolation will end in a finite number of operations. In this particular case, a thread is considered to execute in isolation as long as the other threads do not progress [HLM03, HLMS03]. The authors also developed a Java library based on their approach [HLM06]. The implementation manages the transactional location of memory using a shadow factory pattern. The shadow factory copies each original field into its shadow version when a transaction begins, so that the transaction writes directly on the original fields. If the transaction commits, no conflicts have been detected with other concurrent thread, so the original fields contain the final values. If the transaction aborts, the values in the shadow fields are restored to the original fields to rollback the transaction.

Rudys *et al.* provide a framework for introducing transactional rollback to programs at the language level [RW02]. They use a strict two-phase locking mechanism to prevent conflicts and a lazy shadow copy strategy to manage the memory. When a transaction locks a location for writing, the data structure at that memory location is backed up in a shadow structure that is used to rollback the values on a transaction abort. This particular implementation of software transactional memory for Java uses Java bytecode rewriting to enable the transactional mechanisms.

We can find a different shadow copy approach in the proposal of Nierstrasz *et al.* [RN07, RN09]. All the objects accessed within a transaction are copied in a *working copy* and all the write and read operations are redirected to the copy, so that the transactions does not update the original memory and they can write optimistically on the copy of the objects. Conflicts are checked only before the commits. If no conflict is detected, the working copy can be safely used to update the original objects, otherwise the conflict is resolved by aborting or retrying the transaction. This strategy allows

to save time in case of an abort because it does not need a state restoration. The authors propose a Smalltalk implementation of their transactional mechanism. The tool works as a transparent source-to-source transformation steered by method annotations to control the automatic code transformation.

Noël describes a particular implementation of the shadow copies strategy [Noe10]. Every transaction has its own private map where it stores a copy of all the objects that are involved in the transaction itself. The transactions operate on the objects that are stored in their own private maps. When a transaction commits, its private map is then added to a queue that is shared among all the transactions. When all the transactions that operate on a particular area of memory terminate their execution, the system checks for conflicts by iterating on the shared queue to see if an object has been modified concurrently. If no conflicts are detected, the original objects are then updated accordingly with the changes done in the transactions and the private map is finally removed from the shared queue.

Cachopo *et al.* provide yet another implementation of shadow copy, called *versioned boxes* [CRS06]. A versioned box is a container that keeps the history of all the updates of the value of a specific memory location in a transactional context. When a transaction updates an object, a new local version of the object is created and linked to the versioned box related to that object, so that the history of all the changes is stored in the box. Beside the local values and the link to the boxes, a transaction keeps a set of boxes that were read in the context of the transaction itself. Conflicts are managed at commit time: a transaction commits successfully if none of the boxes that were read during its execution changed after its start, and the final values can be reported in memory. Otherwise, the transaction aborts, it does not have any effect on the versioned boxes and all the local values, if any, are lost.

Software transactional memory, even if it is a promising approach to simplify concurrent access to shared resources, suffers from problems related to performance and impurity of the transactions. Cascaval *et al.* observe that software transactional memory has higher sequential overhead than traditional techniques. This is due to the load and store operations that are expanded within the transactions and because of the additional instructions that constitute the transactional mechanism implementation. The authors also show that, to improve performance, software transactional memory mechanisms typically do not expand load and store operations to non-transactional memory accesses. This results in a more complex semantics and has the effect of weakening the atomicity of the transactions, making them unable to detect conflicts on the edge between transactional and non-transactional memory accesses, thus this introduces the need of a more complicated conflict detection mechanism. The authors observe that the complexity and the overhead introduced by software transactional memory mechanisms limits the productivity gain given by the adoption of those techniques, and that the incentive of a migration to such paradigm is not yet appealing [CBM+08].

Yoo *et al.* analyze the overhead introduced in the execution by the software trans-

actional memory mechanism proposed by Intel [YNW+08]. They find four main performance bottlenecks: the conflict detection mechanism has a too coarse granularity that can result in *false conflicts*; the compilers tend to *over-instrument* memory accesses; the common practice of *privatize* shared objects inside critical sections to access them outside the critical section, which increases the overall overhead because of data races; the costs of the startup and the teardown for short transactions is generally *poor amortized* and the system fails to scale.

Brevnov *et al.* [BDK+08] evaluates the performance of the software transactional memory implementation by Adl-Tabatabai *et al.* [ATLM+06] and compares it to a lock based counterpart. The authors develop a set of workloads made of collection of different known benchmarks. They found that is it fairly quick to modify small concurrent Java programs to use software transactions memory, while larger applications need a non-trivial and challenging redesign. The results of the evaluation show that software transactions memory performs slower than locking code and it often introduces an impressive memory overhead.

Software transactional memory mechanisms provide an abstraction for concurrent data access and developers can benefit from the composability and modularity that software transactional memories provide. Although, they also have been proved to be not particularly performant and to suffer from large overhead. We deal with general and potentially interactive systems, and the overhead, in terms of time and space, should be minimum to preserve the responsiveness of the system. The facilities that software transactional memories provide are particularly useful for new or small existing systems, but major modifications might be necessary to adapt medium or large size systems. In this work we want to automatically enhance existing systems with self-healing capabilities with no limitations to small programs. For these reasons we surveyed another way to assure the consistency of the state of a system, checkpoint and recovery mechanisms.

5.2 Checkpoint and Recovery

Recovering error-free information upon a system failure has been studied since the seventies. The simplest strategy consists in taking a snapshot of the system (saving the state of the system) every fixed amount of time. This strategy is simple and safe, but it incurs in several problems such as the large overhead due to the checkpoint operations and the long, in some worst cases, service interruption due to the long recovery time after a failure. The checkpoint frequency plays a key role in these problems. Many researchers, like Chandy *et al.* [CR72] and Gelenbe *et al.* [Gel79] have studied how to calculate the optimal checkpoint interval that both minimizes the mean overhead of the checkpointing mechanism and maximizes the system availability.

A common way to schedule checkpoints along the execution of the system is to place them periodically in time, at every fixed interval. To calculate the optimum

checkpoint interval that maximizes the system availability, many researchers refer to the time distribution of failures, computed for instance with a Poison function [NvS90, SG95, SKG89, OKFN97, HDO10]. A periodic scheduling of the checkpoints might be not flexible enough for some kind of applications, thus some authors propose a non-periodic scheduling. The scheduling can be driven by the age of the system [DKT02, DOK02] or upon certain events [CSSS09].

Saving a snapshot of the entire system at each checkpoint may lead to space problems. An efficient checkpoint approach consists in saving the information incrementally, for example by either storing only the changes in the state of the system made since the previous checkpoint [EJZ92, WM89], or as soon as an update in the state occurs (copy on write) [EJZ92, WP96] or saving only a partition of the state that is relevant to consistently rollback the system [LF90].

Two common ways to implement incremental checkpoints are *page-based* and *hash-based* approaches. In the page-based approach each memory page of the system is augmented with a dirty bit that indicates if the system has modified that page. At the time of the checkpoint, only dirty pages are saved and their dirty bit is set again to zero. This approach requires the operating system support, and sometimes also hardware support [PBKL95, LTBL97]

The dirty bit approach can also apply to the application level. Lawall *et al.* propose to rely on the developers to write specialized code to optimize checkpoint actions. The developers declares the objects that must be checkpointable, and they write the algorithm that specifies how to save the state and what part of the state of each object must be saved. Checkpointable objects are automatically instrumented with a dirty-bit and the necessary code to manage it [LM00].

With the hash-based approach the memory of the system is divided into blocks. A hash function maps a block of memory to a unique hash value. At the time of taking a checkpoint, the hash of each block is computed and stored. At the next checkpoint, the hash value is computed again for each block, if it differs from the value stored before, it implies that the memory block has been modified and it needs to be saved. In this case, the operating system support is not strictly necessary and the size of the memory blocks affects the performance of the approach [NKHL02, AGGM04].

Code analysis can be also be used to increase the performance of incremental checkpoints. For instance, Plank *et al.* combine data-flow analysis and developers directives to automatically determine the variables that can be excluded from a checkpoint [PBK95], while Cores *et al.* use a live analysis to select those variables that are live at a checkpoint time, thus excluding to save dead variables [CRMG12].

Copy on write is another approach to distribute the checkpoint over the system execution: as soon as a memory location is updated, a new checkpoint is created and the old value of the just updated memory location is saved in the checkpoint. This approach works under the assumption that we can detect a write operation. Researchers proposed several techniques and implementations of this approach, either at

the code machine level and at the programming language level. For example, West *et al.* instrument the *store* operations of RISC code of Sparc architectures to save every update in the memory [WP96]. At the programming language level, Feng *et al.* instrument object-oriented programs to detect all the updates to private object fields. This particular choice of saving only private fields relies on the encapsulation principle [Fen05a, Fen05b].

Copy on write and dirty-bit approaches can be combined. Wang *et al.* propose a technique to checkpoint virtual machine guest systems. At the beginning of a checkpoint interval, the memory of the guest systems is saved in the checkpoint, and all memory pages are set as read-only. When the system writes to a read-only page, it triggers a page fault. The page is then saved into a new checkpoint and it is set as writable [WKII10].

The readers interested in more details about checkpoint mechanisms can refer to the surveys of Roman and Egwutuoha *et al.* [Rom02, ELSC13].

The checkpoint and recovery approaches are usually highly customizable. We can decide where to take a checkpoint and what to save, and this allows us to develop and tune a mechanism with a particular attention on the performance and on the overhead. The checkpoint and recovery mechanisms are also easy to integrate in existing systems as additional component.

5.3 Considerations on the State Consistency Mechanisms

Software transactional memory brings the concept of databases transaction, and some of the ACID properties, into software systems as a new paradigm, opposed to the locking mechanism, to deal with shared memory accesses and concurrent systems. Software transactional memory enables areas of code of the system with automatic rollback capabilities activated by a detector that aborts a transaction when it catches a concurrent access to shared memory.

The checkpoint and recovery approaches provide mechanisms to recurrently save the state of the system and to recover the execution of the system from one of the saved points.

In the approach we presented in Section 3.2, briefly schematized in Figure 3.2 at page 35, we divide the rollback strategy in two phases: the *identification* of an area of code delimited between a failure point and a point where the state of the system can be safely restored, and the *re-execution* of such area of code to exploit the available redundancy. In this context, a mix of the two approaches, software transactional memory and checkpoint and recovery, might be preferred. Checkpoints can be taken systematically along the execution of the system and used to dynamically identify the area of code to re-execute. Such area can, then, be considered a transactional area that must be executed without failures or else all the effects must be avoided.

We explain in details our recovery approach in Section 6 and its prototypal implementation in Section 7.

Chapter 6

Frames

Failures are manifestation of faults. Failures are unexpected behaviours and may manifest as wrong results or corrupted state. A failure may be caused by one or more faults and may manifest at an arbitrary time far from the execution of the fault. Thus, it is not straightforward to understand when the state has been corrupted, and thus, what is the last consistent state of the system on which we can rely to recover the system and find a workaround to avoid the failure. In this chapter we propose the concept of frames to tackle these challenges.

Our approach automatically avoids system failures at runtime by exploiting the intrinsic redundancy of software systems to find an alternative way to provide the same functionality without incurring in a failure. In Chapter 3 we introduced the concept of *frames* that are areas of code with alternative executions. Frames enable failure detection, assure state consistency in case of failures and include the redundancy that might avoid the failure.

Researchers have proposed several approaches and techniques to detect failures. Exceptions [Goo75, PW76, Hoa69] have been combined with assertions automatically derived from the system execution [Ros95, PW09, EPG+07] or from formal specification [Mey92, GMM07]; other researches use models to observe deviation in the behavior of the system [DZM09, LMP08, LMP09, dCBGU11]. In this work we assume the existence of some failure detection mechanisms. Our approach is not bounded to a specific failure detection mechanism. However in the experiments we work mostly with assertions.

Given a failure the corresponding faults can be located anywhere before the failure occurrence. Faults, indeed, might have latency time before they manifest themselves as failure [SL86], and may be difficult to locate them. For this reason many fault diagnosis approaches and techniques have been proposed to find the root cause of a failure [HOB05, DZM09, BMP09, SSG+09, DDZS09]. Fault diagnosis techniques often require a large amount of information and they are time consuming, thus they are not

suitable for runtime use. Our approach based on frames tries to balance precision and speed: we do not try to locate a fault, rather we try to find a workaround for it. We assume that it is more probable that the fault is located close to the failure, thus we start to cope with the code executed right before the failure, and we then increase the area of code under examination, considering larger portions of the system and dealing with faults located far from the failures.

To assure the consistency of the state of the system we have to save the state before the failure occurs. Saving the state only right before a failure may not be enough due to the possible fault latency: the faults causing the failure might be arbitrary distant from the actual failure. One way to address this problem is to regularly save the state, to assure at least one consistent state to recover. The state saving frequency impacts on the overall time and space performance of the approach, and the different approaches presented in Chapter 5 try to find a good balance. We take advantage of the knowledge we have about the intrinsic redundancy of the system to save the state only when we can apply a recovery action that might avoid the failure.

We propose to tackle the problems of the consistency and the recovery of the state, of the fault locality and of the failure avoidance by means of *frames*, a dynamic segmentation of the code of the system. In the next section we present the details of the frames and we discuss how they work.

6.1 The Frame Approach

A frame is a portion of code where the state of the system can be consistently assured and recovered.

To assure the consistency of the state we regularly save it along the execution of the system. Considering the potentially large size of the state of the system, saving the whole memory incurs unacceptable costs. To reduce the costs, our strategy considers two aspects: We save only the portion of the state of the system that has been involved in side effects and those variables required to consistently recover the system; We assure the state consistency only in proximity of potential failures that can be treated successfully with the intrinsic redundancy of the system. For this reason, state consistency and failure detection are linked by means of the knowledge about the intrinsic redundancy of the system. For example, if we have a rewriting rule that involves a portion of the code of the system, we know that such portion is potentially redundant and that, if a failure occurs in that portion of the code, we can rely on some equivalent code to avoid that failure. In this case we save the relevant part of the state of the system before executing that code.

With such strategy that involves the knowledge we have about the redundancy of the system, when a failure occurs we can outline three scenarios:

- The fault is located in the last executed redundant code closest to the failure.

- The fault is located in a previous executed redundant code.
- The fault is located in a portion of non-redundant code.

Figure 6.1 exemplifies the position of some redundant code (R_1 and R_2), saving points (S_1 and S_2) and a failure in the flow of the code. If the fault is located in the code R_1 we fall in the first scenario. We then recover the state of the system at S_1, we substitute the code R_1 according to the rewriting rules and re-start the execution to find a workaround. We call the portion of code between S_1 and the failure point, a *frame*. We can repeat this process depending on how many rewriting rules apply to R_1.

If the fault is located in code R_2, we face the second scenario that shows the dynamism of the framing system. We create a first frame to cover the first scenario, but no rewriting rules for R_1 can avoid the failure. We then recover the state of the system to S_2 and we dynamically create a new frame that spans from S_2 to the failure. We now might have a rewriting rule that applies to R_2 and avoids the failure. This can be repeated up to the beginning of the code execution, thus recovering also from fails extremely far from the failure occurrence.

In the third scenario, the fault may be located somewhere else in the code, that is outside of the code fragments R_1 and R_2. This is the most general case. A faulty statement may produce a wrong state that is not involved in any operation for a long part of the execution. Then, it might be that a portion of code uses the corrupted part of the state and the system fails, while a redundant code of the same failing portion of code might not use that part of the state to compute the result and thus may not fail. We face the third scenario in the same way we face the two previous scenarios: we recover the system at S_1 and S_2 and we apply all the substitutions to R_1 and R_2, respectively, since it might be possible that a change in the code will mask the effects of the fault.

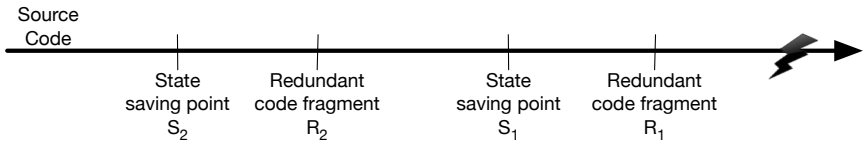


Figure 6.1. Scenario

We assume that the fault is located close to the failing point with high probability, so we first try to recover the execution and exploit the intrinsic redundancy in frames close to the failure. But the assumption that a fault is located near the failure does not always hold because they may manifest after an arbitrary latency, so, as showed in the second scenario, we dynamically resize the frame to deal with the fault latency phenomenon.

The third scenario also shows why we bind the knowledge about the intrinsic redundancy of the system to the frequency of the state saving points. Saving the state with a higher frequency is not necessary because when we consider a larger frame we must assure that we also include new rewriting rules to apply. Recovering the state of the system in between the points S_1 and R_2 does not add any new possibility to avoid the failure.

Frames are distributed everywhere in the code, can be consecutive or overlapping, and can be dynamically activated or deactivated depending on the runtime events. Frames may also have a limited validity in time. They depend on the states of the system saved along the system execution: a frame can only exist if there exists a valid stored state. Thus, the validity of a frame is related to the validity of its recovery point.

Finally, we define a frame as a dynamic triple that includes three components: a state saving point, a portion of redundant code and a failure point. While frames are identified dynamically and online upon a failure event, the other two components (state saving points and portions of redundant code) of the frames are identified statically and offline. The information about the components of the frames collected offline are used to dynamically activate valid frames and find workarounds. In the next sections we discuss the static identification of frame components and their runtime activation.

Static Identification of Frame Components

We identify frame components statically exploiting the knowledge about the redundancy of the system, expressed in the form of rewriting rules. As introduced in Chapter 4, a rewriting rule encodes how a code fragment can be rewritten with different operations without changing the semantics, and it is composed of two patterns, the matching pattern and the substitution pattern. Thus, for each rewriting rule extracted and encoded from the system:

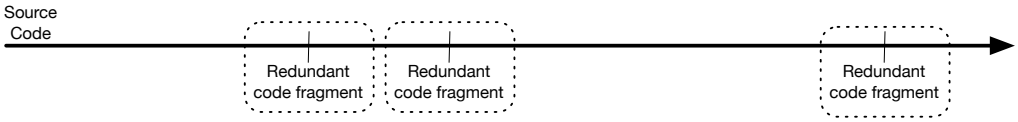
1. We locate the matching pattern of the rewriting rule (Figure 6.2(a)).

2. We place at least one state saving point before the redundant code fragment spotted in the previous step, to enable the state recovery mechanism (Figure 6.2(b)).

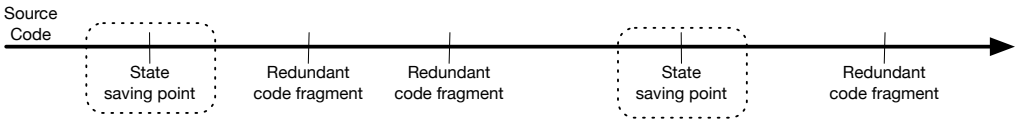
3. We assume a failure detection mechanism that can notify a failure occurrence.

State saving points and the code fragments matched by the rewriting rule are not exclusively related: one state saving point may serve more than one potential failing code fragment.

Figure 6.2 shows the two step of the offline identification of the frame components in a system with three redundant code fragments identified by a list of given rewriting rules.



(a) Locating redundant code fragments



(b) Placing state saving points

Figure 6.2. Frames infrastructure

Dynamic Activation of Frames

We statically build the scaffolding of the frames that are dynamically activated at runtime upon a failure detection. Depending on where the failure event is caught, we can dynamically identify several potential frames. Figure 6.3(a) shows all the potential frames that can be activated at runtime depending on different location of the failure event.

When a failure occurs and it is caught by the failure detector:

1. We dynamically create a frame that includes the code between the failing point and the closest executed state saving point in time (Frame 1 in Figure 6.3(b)). This strategy, which selects the closet saving point, follows the principle of locality: we assume a high probability for the fault to be located close to the failure, so we activate the shortest frame first. The procedure of creating the frames assures that we can apply at least one rewriting rule within the active frame.

2. We recover the state of the system at the point it has been saved and we go back to a consistent state.

3. We select a rewriting rule among the available ones for the current active frame.

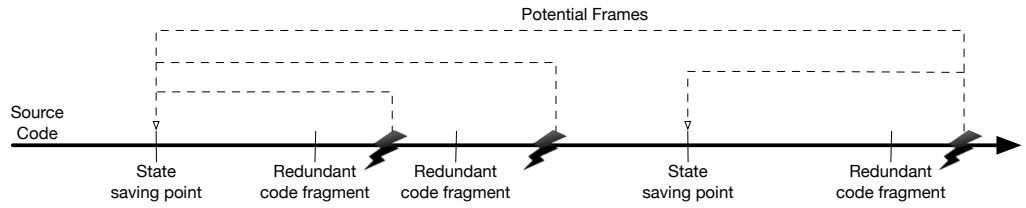
4. We apply the rewriting rule to the code fragment within the current active frame. The result is a different portion of code with the same expected behavior of the one that failed.

5. We execute the new code. If the failure does not occurs again, the system keeps running.

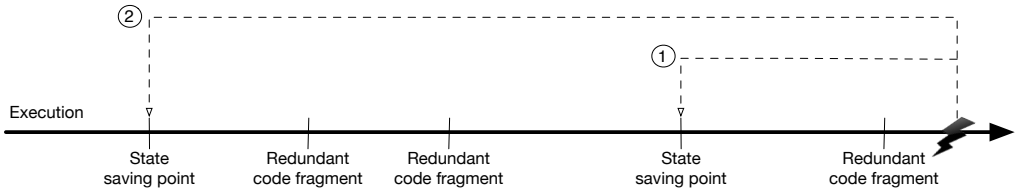
6. If the failure occurs again we apply another rewriting rule, and repeat the process from the step 2.

7. If the system fails after the application of all the rules and there exists at least another previous state saving point in the execution trace, we dynamically activate another frame. We go back on the execution trace to the previous state saving point and enable a new frame: its boundaries are the failing point and the new active state saving point (Frame 2 in Figure 6.3(b)). In a bigger frame we include new redundant code fragments and thus new rewriting rules.

8. We iterate this process until we cannot activate new frames.



(a) Potential frames that can be activated dynamically



(b) Two frames dynamically activated on a single failure point

Figure 6.3. Dynamic frames activation

Rewriting Rules Selection Criteria

Each frame may have several associated rewriting rules. In Chapter 4 we explained the rationale behind the intrinsic redundancy in software systems. In Listing 4.7 at page 46 we show as an example that the method *containsKey* of the class *LinkedHashMap* of the Guava library has four rewriting rules. Each of those rewriting rules may have different probability to avoid a failure. We need a criterion to sort them and select the one with the highest probability to avoid the failure.

We sort rules according to a criterion that takes in consideration the previous success rate of the rewriting rules. The success rate is computed as the fraction of the times the rewriting rule successfully avoided a failure over the total amount of times the rule has been applied. If we know that a certain rewriting rule is more likely to avoid certain kind of failure, we can also adapt this probability by associating the rule

with the failure context, to increase the chance to select a specific rule when a specific failure occurs.

6.2 Originality of Frames

Even though our approach is novel, the idea of organizing redundant code into blocks of code is not new. We can compare our framing system with other approaches in the literature. Randell's Recovery Blocks [Ran75] are redundant blocks of code that are executed sequentially when a failure occurs. The Recovery Blocks technique relies on two main principles: independent implementation and design diversity. Implementing independently the recovery blocks deliberately adds redundancy into the system, while we rely on intrinsic redundancy that results from some development practices and not from an intentional deed by the developers. Moreover, recovery blocks are defined at design time and implemented statically into the system. Given the knowledge of the redundancy in the system, our frames are automatically injected into the system. Our frames are also flexible, they are dynamically activated at runtime and their size is adapted when a frame cannot avoid a failure.

Cabral *et al.* propose another strategy to enclose parts of the system into blocks to provide fault tolerance capabilities called *benign recovery actions* [CM11]. Recovery actions can be specifically developed for the system or they can be general to be easily imported and adapted into different systems, but they are still deliberately developed for reliability purpose. Recovery actions enclose portions of code that might fail in a loop that, in case of failure, select a recovery action to execute before the re-execution of the failed code. Examples of recovery actions are removing temporary files, reducing the size of the swap files, moving files to a remote server or mounting an extra disk. Recovery actions are not then based truly on redundant code, but mostly on reactions to failures that should avoid the failure when the same code is re-executed. Recovery actions can be selected dynamically at runtime, but the blocks of code are still statically defined at design time.

Chapter 7

ARMOR – A Prototype for Java

To evaluate the effectiveness and the performance of our automatic approach to avoid failures at runtime, we designed and developed a Java prototypal tool. The tool adds runtime failure avoidance capabilities to a Java system with respect to a library that the system uses and for which there is an available set of code rewriting rules. The tool instruments an existing Java system and monitors the system for runtime failures transparently to the user. This chapter provides the details of the design and the implementation of our prototypal tool, called ARMOR.

In this chapter we present the architecture of *ARMOR*, the implementation for Java of our technique to automatically recover from failures[1]. ARMOR works in the following general scenario: a Java application fails because of faults in one of the libraries. Such faults may trigger a failure in the library code or in the application code. ARMOR, which is embedded within the application and is notified of the failure, reacts to the failure by first restoring the state of the application to a previously saved state, by means of a set checkpoint, and then by selecting and executing a code rewriting rule that might avoid the failure. If multiple code rewriting rules are available, ARMOR selects one of them based on the selection criterion explained in Section 6.1. The tool iterates this process until it obtains a valid workaround (i.e., a failure-free execution) or until there are no more code rewriting rules left to try. In the first case the execution of the application proceeds as if no failure occurred. In the second case ARMOR forwards the failure (an exception) to the application code as if ARMOR did not exist.

ARMOR works in two phases and with two main components: a preprocessing off-line phase, and a runtime monitoring phase. In the off-line phase, the ARMOR preprocessor analyzes the application to identify where code rewriting rules might be applied and instruments the application with the necessary code to checkpoint and restore the state of the system and to select those alternative sequences of code. At

[1] `http://star.inf.usi.ch/armor/`

runtime, ARMOR records the state of the application at chosen checkpoints that have been set in respect with the code rewriting rules, and then reacts to failures by selecting and activating code rewriting rules. In the next sections we introduce the Java implementation of the concept of the frame and we detail both the preprocessing and the runtime activities.

7.1 Roll-Back Areas – Frames for Java

A roll-back area (RBA) is the primary structural element of the application upon which ARMOR operates. As for *frames*, we define a roll-back area as a segment of the application code within which (1) the application calls one or more operations of any of the libraries, (2) the state of the system can be recovered, and (3) a failure can be detected and reported. A roll-back area should also be *minimal* in the sense that it should be confined to operations that might fail and that could be replaced with a code rewriting rule. This is because the execution of any other code, before or after the library calls, might invalidate the checkpoint (for instance, with I/O operations, see Section 7.3) and in any case would increase its runtime overhead.

A roll-back area shares the same principles of a frame, and it may extend over sections of the application code at any level of granularity, from a single statement to a basic block to an entire method and across methods. If the frames are the general and conceptual idea of enclosing part of the system in respect to some properties, roll-back areas represent their implementation. Our current implementation of ARMOR supports two types of extents for roll-back areas: a whole method body and a single initialization expression for a field (static or not). The extent of a roll-back area is constrained by the mechanism that ARMOR implements to dynamically replace the code of the RBA with one of its pre-compiled variants. This mechanism that we describe in detail in Section 7.3 can replace only entire methods and thus requires RBAs to be encapsulated as a method.

It is also conceivable to encapsulate RBAs consisting of blocks of instructions. However, the encapsulation of such RBAs poses a number of technical problems that we ultimately decided not to address in the scope of this work. Among these problems, the most significant one is the handling of local variables along with the application state that those variables might refer to.

7.2 Preprocessing Phase

The preprocessor starts with the source code of the application, the binary distribution of the libraries, and the specifications of the code rewriting rules for each library. The preprocessor (1) identifies the units of code to which the code rewriting rules might be applied, which we call *roll-back areas (RBAs)*, (2) instruments them with the necessary code to set checkpoints and to react to failures, and (3) compiles and stores the

RBA variants to be used as potential workarounds at runtime. The RBA variants are RBAs where the original code fragment that matches a code rewriting rule has been substituted with the corresponding substitution pattern. The pre-compilation phase of the RBA variants is the strategy we decided to take to keep the runtime overhead low. The operation of applying the code rewriting rules, which are coded as regular expressions, to Java code at runtime is very expensive due to the compilation process thus, we pre-compile all the possible matches of the code rewriting rules for each RBA. We also limit the application of the code rewriting rules to one at time for each RBA: for instance, if we find an RBA with a code fragment that matches with two different code rewriting rules and another code fragment of the same RBA that matches with a different code rewriting rule, we produce three different variants of such RBA.

In the following part of this section we explain the details of the three preprocessing steps by means of an example that is composed of a system that uses a target library for which a set of code rewriting rules have been provided. Listing 7.1 shows a simple application intended to illustrate the preprocessing performed by ARMOR. This application uses the JodaTime library, for which we derived a set of code rewriting rules, to get the instant corresponding to midnight of the current date, and would fail on specific dates and time zones. In May 2011 a developer reported issue n. 3304757 with the JodaTime library[2]. As it turns out, that issue was fixed within a short period of time, but the issue is still interesting because of the nature of the fault and the resulting failure. Issue n. 3304757 reported a failure resulting in an exception when trying to get the instant corresponding to the beginning of the day on certain dates in countries that observe daylight saving time (DST). This failure could not be easily detected during testing since it is triggered only under particular conditions, namely in regions where the DST leap occurs over midnight (for example, America/Sao_Paulo every year, some years in some US regions and some Latin America regions).

Identifying Roll-Back Areas

In the first step of the preprocessing, ARMOR identifies roll-back areas. In the system showed in Listing 7.1, ARMOR would identify three roll-back areas. These are the initialization of field tz on line 2, the initDayAndZone method on line 5, and the setMidnight method on line 12, since they all contain at least one invocation of the JodaTime library. The readers should notice that ARMOR supports nested RBAs, that is, RBAs that invoke other RBAs. Method initDayAndZone is an example of a nested RBA, since it uses JodaTime directly, and invokes setMidnight, which is itself an RBA.

To identify RBAs we statically analyze the bytecode of the system using SOOT[3], a Java optimization framework, which also provides facilities to build and analyze the call graph of a system. With the call graph of the entire system we can identify all the

[2] http://sourceforge.net/tracker/?func=detail&aid=3304757&group_id=97367&atid=617889

[3] http://www.sable.mcgill.ca/soot/

```
1  class CurrentMidnight {
2    DateTimeZone tz = DateTimeZone.forID("America/Sao_Paulo");
3    DateTime midnight;
4
5    public void initDayAndZone(){
6      DateTimeZone.setDefault(tz);
7      DateTime dt = new DateTime();
8      ...
9      setMidnight(dt);
10   }
11
12   private void setMidnight(DateTime dt){
13     midnight = dt.millisOfDay().withMinimumValue();
14   }
15
16   public DateTime getMidnight(){
17     return midnight;
18   }
19 }
20
21 class Main {
22   public static void main(String args[]){
23     ...
24     CurrentMidnight cm = new CurrentMidnight();
25     cm.initDayAndZone();
26     ...
27     cm.getMidnight();
28     ...
29   }
30 }
```

Listing 7.1. Example application code

calls to the target library with little effort.

Once the call the library as been identified, ARMOR ensures that there is at least one code rewriting rule that can be successfully applied to the identified code. A code rewriting rule can be successfully applied when it matches a piece of code of the system and the substitution pattern produces a syntactically correct code. We verify the syntactic correctness by compiling the new code. If there is at least one code rewriting rule that compiles, ARMOR keeps the RBA, otherwise ARMOR discards it because we do not have chance to avoid a failure that can potentially affect that piece of code.

Thus the ARMOR preprocessor identifies RBAs consisting of a method body, which do not need additional encapsulation, and field initialization expressions, which need to be encapsulated through an ad-hoc additional method.

RBA Encapsulation and Proxy Methods

Once the RBAs have been identified, the preprocessor encapsulates and instruments them to allow them to be dynamically replaced with alternative variants at runtime. The encapsulation applies only to RBAs consisting of initialization expressions, which must be rewritten as methods. Let us consider the initialization expression of the field tz. ARMOR creates a new method (called tz_init()) that encapsulates the initialization of the new object and returns it. The semantics of the program remains the same, but now we can treat the initialization expression as a normal class method. Listing 7.2 shows the result of the encapsulation of the initialization for the field tz.

```
1  class CurrentMidnight {
2    DateTimeZone tz = tz_init();
3
4    public DateTimeZone tz_init() {
5      return DateTimeZone.forID("America/Sao_Paulo");
6    }
7
8    // The remaining part of the class untouched
9    ...
10 }
```

Listing 7.2. Encapsulation of an initialization expression

Once each RBA is encapsulated as a method, the preprocessor creates a proxy method for each RBA method. The role of the proxy is to enclose the call to the original RBA method within a loop of operations consisting to set the checkpoint, call the original RBA method and respond to potential failures.

Listing 7.3 shows the proxy methods created for two of the three RBAs identified in the example of Listing 7.1: the proxy method for the encapsulation of the initialization expressions (method tz_init_original on line 3 with its proxy method tz_init on

```
1  class CurrentMidnight {
2   DateTimeZone tz = tz_init();
3   public DateTimeZone tz_init_original() {
4    return DateTimeZone.forID("America/Sao_Paulo");
5   }
6   public DateTimeZone tz_init() {
7    try {
8     create_checkpoint();
9     return tz_init_original();
10   } catch (Exception ex) {
11    while (more_rba_variants_available) {
12     try {
13      restore_checkpoint();
14      load_new_rba_variant();
15      return tz_init_original();
16     } catch (Exception ex1) { // record failure and adjust priorities ... }
17    } throw ex;
18   } finally {
19    discard_checkpoint();
20   }
21  }
22  DateTime midnight;
23  // initDayAndZone proxy method not shown ...
24  public void setMidnight_original(DateTime dt) {
25   midnight = dt.millisOfDay().withMinimumValue();
26  }
27  public void setMidnight(DateTime dt) {
28   try {
29    create_checkpoint();
30    setMidnight_original(dt);
31   } catch (Exception ex) {
32    boolean success = false;
33    while (!success && more_rba_variants_available) {
34     try {
35      restore_checkpoint();
36      load_new_rba_variant();
37      setMidnight_original(dt);
38      success = true;
39     } catch (Exception ex1) { // record failure and adjust priorities ... }
40    }
41    if (!success) throw ex;
42   } finally {
43    discard_checkpoint();
44   }
45  } ...
46 }
```

Listing 7.3. Result of preprocessing (simplified)

line 6) and the proxy method for the method setMidnight (method setMidnight_original on line 24 with its proxy method setMidnight on line 27). For the sake of readability, we omit some details and show a simpler code than the one produced by ARMOR.

Let us now consider the RBA that consists of the original method setMidnight. ARMOR renames this method to setMidnight_original and creates a proxy method called setMidnight with the same signature as the original method. The proxy method declares and handles the same exceptions declared and handled in the original method. In this respect, ARMOR distinguishes between checked and unchecked exceptions. In Java, exceptions can be either checked or unchecked. Checked exceptions are invalid or simply special conditions that are explicitly declared as potential outcomes of the calls to library functions. These are exceptions that the application code deals explicitly, either by handling them or by passing them up in the stack. By contrast, unchecked (or runtime) exceptions are unexpected conditions that may or may not be handled explicitly by the application.

ARMOR ignores checked exceptions, since it is the responsibility of the programmer to handle those, and in many cases those may well represent a normal path of execution for the application. Therefore, masking those exceptions may interfere with the correct behavior of the application. On the other hand, unchecked exceptions typically represent failures, and therefore ARMOR catches them and responds to them. In practice, the proxy method catches all exceptions with a generic catch statement, but must also explicitly catch all the specific exceptions thrown by the original method only to immediately re-throw them to the application.

In addition to handling the exceptions thrown by the original RBA method, the proxy handles the state of the application (and the library) during the execution of the original RBA and of potential workarounds. In particular, the proxy sets a checkpoint for the state of the application immediately before the execution of the RBA itself (line 29). Then, in case of failure, the proxy restores the state to that checkpoint (line 35) before trying an alternative variant. Before terminating, the proxy can discards the checkpoint.

RBA Variants

For each identified roll-back area, the ARMOR preprocessor produces a series of alternative variants of the application code by applying the rewriting rules of each library used within that roll-back area. These are variants of the original application methods as well as of the ad-hoc methods produced by ARMOR to encapsulate initialization expressions. In practice, referring to the example of Listing 7.3, these are all the methods with the _original name suffix. The preprocessor produces one variant for each application of a single rewriting rule. The preprocessor then pre-compiles all the RBA variants and stores the bytecode in a database for potential retrieval and use at runtime.

Let us consider the RBA consisting of the method setMidnight_original (reported in Listing 7.5) and assume that the list of the available code rewriting rules includes the

two rewriting rules listed in Listing 7.4.

$$.millisOfDay().withMinimumValue() \equiv .toDateMidnight().toDateTime()$$
$$.millisOfDay().withMinimumValue() \equiv .withTimeAtStartOfDay()$$

Listing 7.4. Rewriting rules for the JodaTime DateTime class

The RBA setMidnight_original matches both the code rewriting rules, so ARMOR produces two variants (Listing 7.5), one for each rule, pre-compiles and stores them for later use.

Original:

```
1 public void setMidnight_original(DateTime dt) {
2   midnight = dt.millisOfDay().withMinimumValue();
3 }
```

Variant 1:

```
1 public void setMidnight_original(DateTime dt) {
2   midnight = dt.toDateMidnight().toDateTime();
3 }
```

Variant 2:

```
1 public void setMidnight_original(DateTime dt) {
2   midnight = dt.withTimeAtStartOfDay();
3 }
```

Listing 7.5. Original RBA setMidnight with two variants

We are sure that ARMOR can produce at least one valid variant per roll-back area, indeed, those roll-back areas that do not have any applicable rule are filtered out in the first step of the identification. When a code rewriting rule matches a RBA and produces a valid variant, such variant is compiled and the resulting bytecode is stored for run-time use. When a code rewriting rule matches a RBA and produces a non-compilable variant, such variant is then discarded. ARMOR produces non-compilable variants because in the process of rewriting the code with the regular expressions, some contextual information may be lost. For instance, since ARMOR substitutes the code with a string matching strategy, it does not check the type of the objects at code substitution time. Polymorphism or method name similarities can then produce some erroneous matches that lead to produce syntactically correct but semantically wrong code. The pre-compilation process assures that the variants, produced with the code substitution, can run, even though it is not a guarantee that it is an effective workaround.

7.3 Runtime

After the preprocessing phase, the application can be compiled and deployed. ARMOR does not run any special component alongside the application, so the execution of the instrumented application differs from the original one only in the execution of the proxy methods. ARMOR assumes the existence of a failure detector, which may be implemented as a separate autonomous component. However, in our prototype we assume a typical lightweight failure detection based on assertions and runtime exceptions.

In practice, the most significant difference in the execution of the instrumented application is the checkpointing of the application state performed within the proxy methods. In the case of successful execution, this is also the only difference.

Checkpointing and Restoring Application State

ARMOR implements an ad-hoc mechanism to checkpoint and restore the state of the application during execution. The high-level semantics of this mechanism is that of a classic checkpointing mechanism: a checkpoint can be set during the execution of the application, then later the checkpoint may be restored, in which case the state of the application is brought back to what it was at the time the checkpoint was set. The same checkpoint may be restored multiple times.

Since RBAs may be nested and several workarounds might be tried at different levels of the execution of the application, ARMOR maintains a thread-local stack of active checkpoints. Every time a proxy method sets a checkpoint, for instance line 29 in Listing 7.3, ARMOR pushes a new checkpoint handle on the stack.

ARMOR implements two interchangeable types of checkpoints, one based on a snapshot taken before the execution of the RBA code, and one based on a lazy change-log recorded during the execution of the RBA code.

The first mechanism takes a snapshot of the portion of the application state that might be modified by the execution of a roll-back area at the time the checkpoint is set. In its basic form, the snapshot consists of the transitive closure of all the objects reachable from the object on which the RBA method is called (i.e., this object) plus the parameters to the RBA method and all the static fields that are accessible by the RBA method. The second mechanism uses a change-log whereby the first time a field is written (static or not, primitive values as well as references) the previous value of that field is recorded in the change-log, so that it can be restored later.

The two mechanisms have complementary advantages and disadvantages. The snapshot saves every value that is part of the application state and that is accessible by the RBA at the checkpoint, regardless of whether it is actually modified by the execution of the RBA. The change-log saves every value that is actually modified, regardless of whether that value is part of the application state at the checkpoint. So, the snapshot incurs a potentially high cost at the time the checkpoint is set, but then incurs no

cost during the execution of the RBA. Conversely, the change-log incurs no initial cost when the checkpoint is set but may incur a high cost during the execution of the RBA. Both mechanism may be improved through static analysis, although in our current implementation of ARMOR we only applied such analysis to the snapshot method, in order to exclude from the transitive closure those objects that are for sure never modified in the execution of the RBA. In Chapter 8 we analyze the performance of both mechanisms in all our experiments.

Replacing Code

When a failure occurs within an RBA, the proxy method replaces the code of the RBA with a variant of that RBA. ARMOR implements this dynamic code replacement by substituting the code of the whole class that contains the RBA, since this is the only way that code can be dynamically redefined in Java. This is done using the *redefineClasses* method of the *java.lang.instrument* package[4].

ARMOR selects one of the pre-compiled classes produced by the preprocessor for that RBA. Each pre-compiled class is derived from the original instrumented class (with proxy and original methods) with only the original method changed. The effect of reloading such a class is to change the original method of the current RBA, and all the future calls to that method. This redefinition does not affect the execution of the active methods of the class (on the current stack). In particular, it does not affect the execution of the proxy method, which is in fact the one that initiates the class redefinition. In practice, referring to the example of Listing 7.3, after replacing the RBA code (line 36), the call to the RBA method (line 37) will execute the new RBA variant, different from that previously executed (line 30 or line 37).

[4] `http://docs.oracle.com/javase/6/docs/api/java/lang/instrument/Instrumentation.html`

Chapter 8

Evaluation

We evaluated our approach with the ARMOR prototype. We first evaluated our approach qualitatively by applying our tool on three real faults where the workaround was already known, to see if the approach could effectively avoid runtime failures. Then to evaluate the effectiveness and the runtime overhead of the technique, we evaluated the approach quantitatively with seeded fault on four open source applications and two libraries. In this chapter we describe the evaluation process, we report and then discuss the results of the experiments. The results show that our approach is effective, that the tool we developed has a limited overhead, mostly related to technological problems.

The research hypothesis of this dissertation, as state in Chapter 1, is that *"software systems are intrinsic redundant, that is they inherently provide multiple ways to perform the same operations. This redundancy can be captured, expressed and used for avoiding failures automatically and at runtime in common use software systems"*. In Chapter 4 we provided some qualitative and quantitative evidences that common software systems are indeed intrinsic redundant and how we can capture and represent this redundancy. Chapters 3, 6 and 7 describe our approach, *Java Automatic Workaround*, which exploits the intrinsic redundancy to avoid failures at runtime and our tool that implements the technique.

To validate our research hypothesis that redundancy can be used to deal with software system failures at runtime, we studied the effectiveness of our approach in avoiding failures and the efficiency of our tool to make it usable for common use software systems. The research questions that we want to address with the evaluation are:

Q1 Can we use workarounds to cope with runtime failures in common systems?

Q2 Can the intrinsic redundancy present in common software system effectively make such systems more resilient to faults?

Q3 Can we use the intrinsic redundancy efficiently?

The first question (Q1) explores the feasibility of our approach in successfully avoiding failures, that is to recover a system after a runtime failure, use a workaround to avoid it, and let the execution of the system continue normally. To answer to this question we surveyed the failure repository of the Java library JodaTime and we selected and analyzed three real faults. This analysis shows the feasibility of the approach, but not its actual effectiveness, that is the focus of the second question (Q2). Given the evidence that we can avoid a failure using a workaround, we evaluated how much the approach is effective by injecting a large amount of faults into a system and applying our tool ARMOR to see how many of them can be avoided automatically. We want to deal with runtime failures in common systems, which can be batch or interactive systems. In both cases, we have not only to avoid failures automatically, but the process should be transparent to the user and efficient enough to preserve the usability of the system. Thus, to answer to the third question (Q3) we measure the performance of our tool ARMOR.

In this chapter we explore the three questions by first giving an overview of the systems we used, then we answer to the first question describing a qualitative experiment on three case studies with real faults. We then answer the second and the third question, thus we introduce the evaluation process for a quantitative experiment and we analyze the results. Finally we discuss the limitation and the threats to validity.

8.1 Applications

Our tool ARMOR works in the scenario where a system fails because the incorrect interaction with a library, and in Chapter 4 we presented the investigation we made on four popular Java libraries, Guava, JodaTime, SWT and Lucene, to identify the amount of redundancy present. For this evaluation we selected two of them: Guava and JodaTime. From these libraries, we selected four applications that use them. The four applications are:

- *Fb2pdf:*[1] a command-line utility to convert files from the FB2 e-book format into PDF. Fb2pdf originally uses the standard Java date/time library, but we modified the application to use the fully compatible JodaTime library and we assured the same original behavior also with the new changed library.

- *Carrot2:*[2] an open source search results clustering engine. It organizes collections of documents into thematic categories. Carrot2 uses the Guava library.

[1] `http://code.google.com/p/fb2pdf/`
[2] `http://project.carrot2.org`

- *Caliper:*[3] a framework for writing, running and viewing the results of Java microbenchmarks and more general code measurements including memory allocation and consumption. Caliper uses the Guava library.
- *Closure:*[4] a source-to-source optimizing JavaScript compiler. It parses JavaScript files, analyzes it, rewrites it and minimizes it. It checks syntax, variable references, and types. Closure uses the Guava library.

As a first step in our experiments, we wrote the rewriting rules for Guava and JodaTime based on their respective API documentation. The result of the study produced 1715 rewriting rules for Guava and 135 for JodaTime. The study is summarized in Table 4.1 in Chapter 4.

Then, focusing on the relevant equivalences for our experiments, we abstracted and formalized those equivalences through code rewriting rules. Starting from the equivalences we extracted from study described in Chapter 4 for the libraries Guava and JodaTime, we wrote 63 code rewriting rules for Guava and 100 for JodaTime out of the 1715 and 135 rewriting rules, respectively. Rewriting rules may specify particular conditions, such as types or the applicability context, that may lead to a proliferation of rules, but many rewriting rules can be synthesized with only one code rewriting rule because such information are abstracted. The code rewriting rules we wrote for the libraries are listed in Appendix A.

8.2 Real Faults in JodaTime

To demonstrate that our approach can use workarounds to cope with runtime failures in common use systems, we experimented with three case studies taken from three real fault reports from the issue repository of the Java library JodaTime.

Issue n. 1375249 A developer reports that if a YearMonthDay object, that is a JodaTime class that supports the year, the month of the year and the day of the month fields, is created with a Calendar (the java.util.Calendar standard Java class) as a parameter, the method plusDays(), which is part of the class YearMonthDay and returns a copy of the date plus the specified number of days, throws an IllegalArgumentException when the resulting date is in the next year[5]. Listing 8.1 shows a small portion of code that reproduces the failure. The small program fails at line 3 with the message "java.lang.IllegalArgumentException: Fields invalid for add".

The developer also provides a workaround for this fault: if the YearMonthDay object is constructed by explicitly specifying an ISOChronology calendar as parameter, the code does not fail.

[3] http://code.google.com/p/caliper/

[4] http://code.google.com/p/closure-compiler/

[5] http://sourceforge.net/tracker/?func=detail&aid=1375249&group_id=97367&atid=617889

```
1 Calendar calendar = new GregorianCalendar(2005, 12, 5);
2 YearMonthDay ymd = new YearMonthDay(calendar);
3 YearMonthDay ymd1 = ymd.plusDays(28);
```

Listing 8.1. Code that reproduces the issue n. 1375249 in JodaTime

$$Class\ YearMonthDay :$$
$$new\ YearMonthDay(\$P_1) \equiv new\ YearMonthDay(\$P_1,\ ISOChronology)$$

Listing 8.2. Rewriting Rule for YearMonthDay in JodaTime

From the hint provided by the developer, we could write a rewriting rule that enclose the knowledge to reproduce the workaround (see Listing 8.2) and the corresponding code rewriting rule (see Listing 8.3)

$$new\ YearMonthDay((?:\backslash s*)([a-zA-Z0-9\_.()"]*[^\ ])(?:\backslash s*))$$
$$\equiv$$
$$new\ YearMonthDay(\$1, ISOChronology.getInstance())$$

Listing 8.3. Code Rewriting Rule for YearMonthDay in JodaTime

We developed a small program that includes the failing statements in Listing 8.1 and run ARMOR on it. Our tool instrumented the system correctly, identifying as rollback area the portion of code of the system the refers to the library JodaTime, and produced the rollback area variant, like the one proposed by the developer as workaround, that overcomes the failure at runtime (see Listing 8.4).

At runtime, ARMOR can detect the failure through the exception handler, recover a consistent state of the system after the failure and substitute the code of the rollback area to overcome the failure.

Issue n. 3072758 A developer reports that the method parseDateTime of the class DateTimeFormatter (a JodaTime class) fails to parse a DST leaping date even if the LenientChronology is specified. A LenientChronology should be tolerant to DST leaps, so no exception should be raised in this case[6]. Listing 8.5 shows few lines of code that exercise the bug.

At line 3 of Listing 8.5 the system fails to parse the date and it raises the follow-

[6] http://sourceforge.net/tracker/?func=detail&aid=3072758&group_id=97367&atid=617889

```
1 Calendar calendar = new GregorianCalendar(2005, 12, 5);
2 YearMonthDay ymd = new YearMonthDay(calendar, new ISOChronology);
3 YearMonthDay ymd1 = ymd.plusDays(28);
```

Listing 8.4. Workaround that fixes the code in YearMonthDay

```
1 DateTimeFormatter formatter = DateTimeFormat.forPattern("dd.MM.yy HH:mm:ss").↩
      withLocale(Locale.getDefault()).withZone(DateTimeZone.forTimeZone(TimeZone.↩
      getDefault()));
2 formatter = formatter.withChronology(LenientChronology.getInstance(ISOChronology↩
      .getInstance(DateTimeZone.forTimeZone(TimeZone.getDefault()))));
3 formatter.parseDateTime("28.03.2004 02:15:00");
```

Listing 8.5. Code that reproduces the issue n. 3072758 in JodaTime

ing exception: "java.lang.IllegalArgumentException: Cannot parse '28.03.2004 02:15:00': Illegal instant due to time zone offset transition (Europe/Berlin)".

$$Class\ DateTimeFormatter:$$
$$.parseDateTime(\$P_1) \equiv .parseLocalDateTime(\$P_1)$$

Listing 8.6. Rewriting Rule for parseDateTime in JodaTime

The developer also provides a workaround to avoid this failing behavior: using the method parseLocalDateTime (note the *Local*) instead of parseDateTime avoids the issue. Thus, we formalize this knowledge into a rewriting rule (see Listing 8.6). To work in ARMOR, the rewriting rule must be coded into a code rewriting rule, as shown in Listing 8.7.

ARMOR successfully instruments the system given as example in Listing 8.5 by identifying the call to the JodaTime library and producing the code variant that can avoid the failure (Listing 8.8). At runtime, ARMOR captures the exception, recovers the state of the system after the failure, and replaces dynamically the code variant to avoid the failure and continue the execution of the system.

Issue n. 3304757 A developer reports a failure resulting in an exception when trying to get the instant corresponding to the beginning of the day on certain dates in countries that observe daylight saving time (DST)[7]. We explain this issue in Chapter 7, where we also provide the knowledge given by the developers, we describe how we

[7] http://sourceforge.net/tracker/?func=detail&aid=3304757&group_id=97367&atid=617889

$$.parseDateTime((?:\backslash s*)([a-zA-Z0-9\_.()+:"]*[^\ ])(?:\backslash s*))$$
$$\equiv$$
$$.parseLocalDateTime(\$1)$$

Listing 8.7. Code Rewriting Rule for parseDateTime in JodaTime

```
1 DateTimeFormatter formatter = DateTimeFormat.forPattern("dd.MM.yy HH:mm:ss").↩
      withLocale(Locale.getDefault()).withZone(DateTimeZone.forTimeZone(TimeZone.↩
      getDefault()));
2 formatter = formatter.withChronology(LenientChronology.getInstance(ISOChronology↩
      .getInstance(DateTimeZone.forTimeZone(TimeZone.getDefault()))));
3 formatter.parseLocalDateTime("28.03.2004 02:15:00");
```

Listing 8.8. Workaround that fixes the code in ParseDateTime

coded it into rewriting rules and code rewriting rules, and how ARMOR works to recover and avoid the documented failure at runtime.

In these three experiments we analyzed the issue reports to understand the faults, the runtime failures, and the workarounds proposed by the developers. To conduct the study, we then coded the knowledge about the workarounds into rewriting rules and then code rewriting rules to employ our tool ARMOR, which resulted successful in all the three cases. The three case studies described above demonstrate that our approach can, indeed, automatically generate workarounds that avoid failures.

8.3 Mutation Analysis

To demonstrate that our approach and our tool ARMOR can effectively exploit the intrinsic redundancy of common systems to avoid functional failures automatically and at runtime, we need to obtain an extensive coverage of the features of the library introduced above, to also obtain more statistically significant results. Real faults taken from fault trackers are often difficult and expensive to reproduce in large quantity. Thus, we apply mutation analysis to introduce a large amount of faults into the three libraries. A study by Andrews *et al.* [ABL05] suggests that mutation faults can in fact be representative of real faults.

As a systematic approach, we proceeded as follows:

1. We used the *Major* mutation analysis framework [JSK11] to inject faults in the libraries. Major is a mutant injector integrated into a Java compiler. It uses conditional expressions to encapsulate the mutants and the original version of

the program in the same basic block, so that each single mutant can be triggered by enabling its identifier at runtime. This strategy assures that a mutant that is not reached cannot be executed under any circumstance. The operator groups available in Major includes replacement of unary and binary arithmetic, logical, relational and shift operators; replacement of a literal value by a positive value, a negative value, and zero.

2. We then ran all the applications with the mutated versions of their respective libraries but we no mutants active, and we traced those executions. For each application we obtained an input that we deemed representative for the application. For Fb2pdf we used a third-party e-book file, for Carrot2 and Caliper we used inputs provided by the developers for demonstration purposes, for Closure we used a large and popular JavaScript library (jQuery). We assumed that the results of the execution correspond to the expected result and we used as an oracle in the rest of the experiment to discern correct and incorrect results. Based on the execution traces, we also discarded the mutants that were never executed.

3. We activated each remaining mutant individually and executed all applications in the presence of each mutant. For each application and mutant, we observed and categorized the outcome of the execution as *error*, *loop*, and *success*, when the execution led to an error (if the outcome differs from the expected one) or exception, an infinite loop, or a normal termination, respectively. We further analyzed the mutants in the *success* category to distinguish mutants whose execution produced the expected output, which we classified as *equivalent* and discarded, and mutants whose execution failed to produce the expected result, which we classified as *non-equivalent*.

4. We then executed the applications instrumented with ARMOR on all *error*, *loop*, and *non-equivalent success* mutants and we measured how many times we could overcome the failure raised by the mutant. For the *error* mutants we simply relied on the implicit failure detection (i.e., exceptions). For the mutants in the *loop* and *non-equivalent success* categories, we augmented the application with specific failure detectors that we obtained as follows:

    - We used Daikon [EPG+07] to derive invariants from repeated executions of the original program (without mutations).
    - We used Daikon on each mutant (same application, same input) and selected those invariants found within roll-back areas that were valid for the original program but not for the mutant program.
    - We inserted those invariants as assertions in the application code, within the RBA where they were found.

Applying ARMOR on the applications

This section presents the result of the instrumentation of the four systems by our tool ARMOR. The results are reported in Table 8.1.

Given a set of code rewriting rules for each library, Table 8.1 shows how many methods of the applications contain a method call to the library that matches one of the code rewriting rules (*RBAs found*). We considered only the RBAs that can produce at least one variant (*RBAs with variants*). The reason is that not all the method calls that match a code rewriting rule produce valid variants. Only the substitutions that produce a syntactical valid code, that is a code that compiles, are kept (*Generated variants*). Note that some RBAs have only one variant, some other have more than one (*Average variants per RBA*).

	Caliper	*Carrot2*	*Closure*	*Fb2pdf*
Code Rewriting Rules	63	63	63	100
RBAs found	130	139	2099	17
RBAs with variants	60	106	687	17
Generated variants	86	191	996	53
Average variants per RBA	1.43	1.80	1.45	3.12

Table 8.1. Results of the preprocessing on the applications

Effectiveness

We evaluate the effectiveness of ARMOR running the experimental process we described in Section 8.3 on the four applications introduced in Section 8.1: Caliper, Carrot2, Closure and Fb2pdf. We measure the effectiveness by counting the cases in which ARMOR could recover from one or more failures caused by a mutant and allows the applications to run to completion with a correct output with the selected test suite.

Table 8.2 summarizes the selection and classification of mutants, explained in Step 3 of Section 8.3, for each selected application.

The results of the experiment are displayed in Table 8.3. The mutants executed with ARMOR are the sum of the mutants that produce an error or an exception, those that do not let the test suite to terminate and those that produce a different output, respectively, the rows *error*, *detected loop* and *detected non-equivalent success* in Table 8.2. We considered ARMOR successful when the program, despite the presence of a failure-inducing fault, terminated *completely* successful; that is all the exceptions due to the mutant are successfully handled and the output corresponds to the expected output of the test suite. These results are very encouraging, since they demonstrate that ARMOR is successful with between 19% and 48% of the mutants.

For each mutant and each application, we manually analyzed the results of the

				Caliper	*Carrot2*	*Closure*	*Fb2pdf*
Generated Mutants				21297	21297	21297	16858
Mutants executed by the test suite				309	187	344	2200
execution	*success*	*equivalent*		210	120	177	1805
		non-equivalent	*detected*	0	2	0	0
			not detected	0	8	3	1
	loop	*detected*		0	1	0	0
		not detected		12	9	15	47
	error			87	47	149	347

Table 8.2. Classification and selection of mutants

	Caliper	*Carrot2*	*Closure*	*Fb2pdf*
Mutants executed with ARMOR Detected non-equivalent success + detected loop + error	87	50	149	347
Mutants where ARMOR is successful	24	24	70	67
Success percentage	28%	48%	47%	19%

Table 8.3. Effectiveness of ARMOR

execution with ARMOR to identify the causes of the successes and failures of our technique.

An interesting case is Fb2pdf. Fb2pdf is the application with the fewest number of RBAs, only seventeen compared to an average of almost eight hundreds for the other three applications, but also with the largest set of mutants affecting the execution, more than two thousands compared to the about three hundreds mutants executed by the other three applications. Fb2pdf is also the application with the lowest success rate, nineteen percent. Analyzing the execution trace, we found that in fact Fb2pdf uses the library quite extensively, but does it through few access points and this explains the low amount of RBAs. Moreover, the application exploits the library at a greater depth than the other applications, and this justifies the large amount of mutants executed by the test suite. We draw two conclusions from this analysis: first, there is little hope to avoid the effects of a fault whenever a few calls use a large portion of the library code, since that would require a large amount of redundancy. Second, workarounds are likely to be more effective when the fault (in the library) is somehow *closer* to the interface, thus, to the application code, and therefore when the alternative use of the library would have a more direct control in steering the execution away from the fault.

Another interesting and related case is that of a failure in Carrot2 which ARMOR could not avoid. This failure is caused by a use of the mutated Guava library from within another library used by Carrot2. This means that ARMOR was not involved in that particular use, because we only instrumented the application Carrot2 itself, not

all the included libraries. We do not know whether ARMOR could have prevented the failure, but once again we observe that faults at a greater depth in the call stack have less chances of being avoided through workarounds at the interface between application and library.

The case of Carrot2 is also interesting because it is characterized by several active and also nested RBAs. One of the RBAs of Carrot2 is in fact in its *main* method, and several others are nested up to a depth of 7. Nested RBAs are expensive because they involve more checkpoints and also because they might induce several nested iterations to look for a valid workaround, especially when no workarounds are found for lower-level RBAs. This complexity might explain the overhead incurred by ARMOR with Carrot2.

Finally, we augmented the application with specific invariants to detect the failures in those cases where the system could not either run to completion (*loop*) or terminate with no exceptions but with a wrong result (*non-equivalent success*). It is interesting to realize that with this additional step of the experiment we could detect only 3 more failures out of 98, as shown in the Table 8.2. We noticed that the set of invariants generated running the original system was very close to the set of invariants generated running the mutated system. This is ascribable to the fact that the changes in the behavior produced by the injected mutants were to small to allow Daikon to generate effective invariants.

Runtime Overhead

We evaluate the efficiency of the tool by measuring the runtime overhead of ARMOR. We verify that the execution of an instrumented application would not suffer an unreasonable penalty due to the instrumentation. We measure the overhead of ARMOR in terms of execution time and in terms of allocated memory in a normal non-failing run (a run with no active mutants and that run to completion correctly) on the test suite, and we compare those measurements with the execution of the original application code on the same test suite.

The time overhead introduced by ARMOR can be caused by three components: the failure detection mechanism, the state recovery mechanism, and the runtime code replacement mechanism. The implementation of the three components is detailed in Chapter 7. The results tell us that the overhead introduced by the runtime code replacement mechanism is negligible, so we only assessed the failure detection and the checkpoint and recovery mechanisms.

The first part of Table 8.4 summarizes the results of the analysis on the time overhead. The original running time of the application is compared, at first, with the running time of the instrumented application with the check-pointing system disabled (*Exception-handling only*). In this case the instrumented application differs from the original by only the exception-handling mechanism, which is built with Java try-catch

		Caliper	*Carrot2*	*Closure*	*Fb2pdf*
Time (seconds)	Original application running time	30.13	2.43	5.40	2.26
	Exception-handling only	30.41 (1%)	4.15 (69%)	10.53 (95%)	3.79 (68%)
	Snapshot-based checkpoints	31.78 (5%)	5.32 (117%)	>1h	4.99 (121%)
	Change-log-based checkpoints	30.87 (2%)	4.75 (94%)	15.90 (194%)	4.70 (114%)
Memory (MB)	Original application memory allocation	1.40	8.87	30.56	17.90
	Snapshot-based checkpoints	12.30 (779%)	23.78 (168%)	—	90.94 (408%)
	Change-log-based checkpoints	10.18 (627%)	11.37 (28%)	120.58 (295%)	25.93 (45%)
Recorded checkpoints (approx.)		30	2,350	1,255,000	4
Values saved in change-log-based checkpoints (approx.)		26,000	270,000	1,880,000	9,000

Table 8.4. Overhead incurred by ARMOR in normal non-failing executions (median over 10 runs)

blocks. Then we compare the original running time with the running time of the applications instrumented with the two different implementations of the checkpoint and recovery mechanisms: a *snapshot-based checkpoint* and a *change-log-based checkpoint*.

The results show that, as expected, the change-log-based checkpoint strategy performs better that the snapshot-based checkpoint strategy. Interesting is the case of Closure that was not able to run to completion with the snapshot-based checkpoint mechanism within one hour due to the large amount of checkpoints recorded during the execution. The results also demonstrate that ARMOR incurs a noticeable but also seemingly reasonable overhead with the change-log-based checkpoint, in all cases. In particular, the running time overhead ranges from 2% to 194%.

Interestingly, we initially assumed that the runtime overhead would be attributable to the checkpoint mechanism, since that is essentially the only active code executed by ARMOR in normal non-failing runs. However, a further analysis shows that a significant portion of the total overhead is instead due to the instrumentation alone, which in practice consists of the time needed to execute a try-block in the proxy method. The results show that this overhead counts for more that half of the total overhead.

The somewhat extreme case of the Closure compiler in which RBAs are executed in very hot loops, as evidenced by the high number of recorded checkpoints, also shows that the checkpoint mechanism is quite efficient, since the execution of over 1.2 million checkpoints (with a total of over 1.8 million saved values), incurs only a relatively low 99% overhead (excluding the overhead of the exception-handling mechanisms), corresponding to a bit more than 5 seconds of execution time (on a 2.53GHz Intel Xeon E5630 CPU).

The second part of Table 8.4 confirms that the change-log-based checkpoint performs better than the snapshot-based checkpoint also in terms of memory allocation in all cases. The extreme case of Fb2pdf shows the large difference of the two checkpointing approaches: with only four checkpoints, the change-log-based checkpoint outperforms the snapshot-based one of almost one fourth.

8.4 Discussion

Differently from other approaches, JAW aims to temporary mask the effects of a failure instead of fixing the fault permanently. This allows the technique to automatically and quickly respond to a runtime failure and keep the system running to successfully achieve the users' requests. The study about the three real case studies, presented in Section 8.2, demonstrates that workarounds are indeed a valuable way to address problems. In those three cases the developers were aware of the problem and of the possible workaround, and our technique successfully avoided the problem transparently to the user, automatically and at runtime. In Section 8.3 we validate our technique and we demonstrate its effectiveness by injecting faults into medium size well known and used libraries. As the results show, we could avoid between the 19% and

the 48% of the failures and run the systems to a correct completion. The runtime cost of JAW is also limited: the performance test shows that, in terms of time, our tool ARMOR adds between the 2% and the 194% of overhead time, but also reveals that half of the overhead is due to the Java exception-handling mechanisms.

8.5 Limitations and Threats to Validity

The main limitation of JAW is our hypothesis that users or developers would be able to correctly identify the intrinsic redundancy and encode it into code rewriting rules, which we could not confirmed with a proper experiment.

The tool ARMOR is limited primarily by technological limitation of the Java language. The roll-back areas are an implementation for Java of the concept of frame, and are not that flexible: we bounded the size of the RBAs to the granularity of methods, therefore, the active frame cannot span everywhere in the code (as stated in Chapter 6), but it is delimited by those region in the code identified as RBAs. Moreover, due to another technological limitation of Java, we can dynamically resize the frames only among RBAs that are still on the JVM activation stack.

Another limitation of ARMOR depends on the implementation of the RBA variants. RBAs variants are pre-compiled at instrumentation time and each variant contains one rewriting rule. Therefore, if different rewriting rules can be applied to different statements of the same RBA, those variants cannot be activated simultaneously and we can only have one single point of variation currently active.

Our implementation of the checkpoint and recovery mechanism suffers from two main limitations: it does not deal with I/O and concurrency. I/O operations might invalidate a checkpoint in some cases. If the Java system reads or writes from or to a network stream, the information in the channel cannot be saved and will be lost. Similarly if the Java system writes on a local file, our checkpoint mechanism cannot rollback the content of the file and this might affect the rest of the execution. This limitations might affect correct operation of ARMOR in recovering an execution after a detected failure. Differently, this issue should not affect the case when the system reads from a local file using the facilities provided by the Java framework, in fact we are able to restore all the fields (public and private) of Java objects, including file pointer fields.

The limitation of the experimental work is in the instrumentation process. We extract the information about the intrinsic redundancy from a target library and then we instrument the application that uses such library. The application is usually part of a larger system that includes the Java application itself and several libraries. Thus, it might be the case that a system failure is induced by one of the many libraries in the system that might be caused by the target library. This is a limitation of the actual implementation of the tool, not of the methodology itself. We gathered, coded, and used the intrinsic redundancy of the libraries to instrument the boundaries between the application and the libraries themself. Following the same principle, we can collect

the intrinsic redundancy of the application and of all the components and libraries integrated in the system and recursively harmonize the whole system.

Chapter 9

Conclusions

This thesis explores software redundancy to increase the reliability of software systems. In particular, we propose to exploit a specific type of redundancy that we call intrinsic redundancy. Intrinsic redundancy is naturally present in software systems for several reasons. A newly upgraded system should often guarantee a certain grade of backward compatibility with older systems thus, for example in the case that an updated functionality might require a new interface to work, the same old functionality is often kept to assure the correct operation of older systems. Systems, especially frameworks, are often developed to provide a wide range of functionalities to serve a large variety of different other systems. To increase the portability and the reusability of such systems, functionalities are often duplicated and tailored for different needs. The proliferation of third-party components and libraries also lead to a growth of redundancy in software systems. Many systems integrates several similar third-party components that often provide large sets of overlapped functionalities. Yet another reason relies on the multiple implementation of the same functionalities that have to meet different non-functional requirements.

All of the above provided reasons induce the developers to introduce redundancy in their systems. This redundancy can be found at the interface level, where for example a component might expose the same functionality through multiple equivalent methods, and this often leads to a redundancy at the implementation level, which means that the same functionality is also implemented, thus executed, with different code. This kind of redundancy, that is intrinsically present in software systems, can be exploited at low cost to avoid failures at runtime.

This thesis introduces an approach, JAW, that exploits the intrinsic redundancy, automatically and at runtime, to find workarounds to avoid failures. The idea of the approach is to monitor the system for a failure, recover it to a consistent state to discard all the side effects that the failure might have produced and select a possible sequence of operations that is equivalent in the intent to the failed operations, but that does not fail.

We implemented a prototype, ARMOR, to demonstrate that intrinsic redundancy can be effectively used to avoid failures automatically and at runtime in common use software systems. We tested the prototype on three real bugs and on injected bugs on four medium size applications, showing that the approach is effective and introduces a small overhead.

9.1 Contributions

The main contribution of the thesis is to provide an approach for augmenting systems with self-healing capabilities. The aim is to increase the reliability of software systems by exploiting their intrinsic redundancy. The approach automatically recovers the systems from failures and avoids them by finding an alternative way to achieve the requested and failed functionality. We now summarize in more detail some aspects of the contributions:

- **A fully automated mechanism to augment software systems with self-healing capabilities.** The mechanism, that implements our approach called *Java Automatic Workaround* (JAW), enables general systems to automatically avoid functional failures at runtime. An augmented system can automatically recover from a failure and find a way, at runtime, to restore the expected behavior by exploiting its intrinsic redundancy.

- **A qualitative and quantitative analysis of the potential use of our approach due to the intrinsic redundancy of software systems.** We give some qualitative arguments to support the thesis that intrinsic redundancy is present in modern software systems, and we quantitative analyzed four large, well known and used Java libraries. We informally express the knowledge of the intrinsic redundancy as *rewriting rules*. We also formulate a syntax and a semantics, called *code rewriting rules*, to code the rewriting rules and make them available to use at runtime.

- **An experimental evaluation to show the effectiveness of JAW.** We perform the evaluation with a prototypal tool for Java systems, called ARMOR. Given a set of code rewriting rules for a target library, ARMOR instruments the systems to enable the approach and then, at runtime, it catches failures, recovers the state of the system and finally finds and applies an equivalence to avoid the failure. We evaluate JAW on four Java applications using the intrinsic redundancy extracted from two libraries. The results of the evaluation show that our approach is effective in avoiding a large number of failures and that the prototype, even with some limitations, performs well and it adds a small overhead to the execution.

9.2 Future Directions

The work presented in this dissertation opens problems and ideas for future research. We introduce possible future directions to improve the effectiveness of the approach, and some long-term developments.

- **Improving the failure detection mechanism.** The failure detection mechanisms we use are based on assertions. Even though such mechanisms are interesting, they have some limitations. We are interested in more efficient detection mechanisms that can detect a failure earlier in the execution or give us more information about the fault location to help our approach to find a workaround faster.

- **Fault removal support.** Our approach prevents the system to halt when a fault leads the system to a failure. JAW provides a quick and responsive workaround to fix the problem at runtime by switching the execution to some portion of code that does not fail. Thus, we do not provide a final patch for the fault, but only a temporary fix. We believe that, while we recover and heal a failing execution, we can collect many useful information about the failure and the workaround. It is interesting to understand how this information can improve the ultimate fault removal process.

- **Automatic identification of intrinsic redundancy.** This thesis assumes that the knowledge of the intrinsic redundancy is available and expressed though a set of code rewriting rules. In this work, the process of discovering and collecting the intrinsic redundancy and coding it into code rewriting rules has been carried manually. Even if such process can be done with a reasonable effort by the developers, an automatic or semi-automatic identification of intrinsic redundancy and generation of code rewriting rules is desirable.

- **Exhaustive extraction of the intrinsic redundancy.** The manual process of discovering, collecting, and expressing the intrinsic redundancy of a system requires, as we have shown in this thesis, little effort but it also results in a rather small, yet effective, set of rules. Extracting the intrinsic redundancy extensively, with the support of an automatic or a semi-automatic technique, we could show interesting information such as the degree of coverage of intrinsic redundancy in general systems or the actual coverage of the intrinsic redundancy as represented by the code rewriting rules. Correlating such information would give a better indication on the general probability for a defect or a random failure to have a workaround and thus, we could also provide an estimation of the chances of success for a given system.

- **Design for self-healing.** This work proposes to augment software systems by means of a skeleton of code that enables self-healing capabilities. A long-term

plan is to propose a set of guiding principles on how to design software systems that enclose a self-healing behavior. The idea is to keep the redundancy as a key element in the design and develop a set of design principles around the notion of redundancy. We then propose to include such design principles at the programming language level by extending an existing language with ad-hoc self-healing constructs or proposing a new programming language.

Appendices

Appendix A

List of all the Code Rewriting Rules

In this Appendix, we present the entire set of code rewriting rules of the three Java libraries that we used in our evaluation. In total, we compiled 63 code rewriting rules for Guava and 100 for JodaTime.

Each rule is written using the Java regular expressions syntax[1] and it follows the following convention: ⟨patternToBeReplaced⟩%⟨replacement⟩, where ⟨patternToBeReplaced⟩ indicates the pattern that has to be found and replaced in the code of the system when the rule is applied, and ⟨replacement⟩ indicates the new code that is used to create a new variant.

A.1 Code Rewriting Rules for Guava

```
1  \.contains\(([\S ]*)\)%.count($1) > 0
2  (.*)\.contains\(([\S ]*)\)\)%$1.count($1) > 0)
3  ([\S]*)\.contains\((?: *)([\S &&[^\&*]]*)(?: *)\)%($1.count($2) > 0)
4  \.contains\((?: *)([\S &&[^\&*]]*)(?: *)\)%.elementSet().contains($1)
5  (.*)\.contains\((?: *)([\S &&[^\&*]]*)(?: *)\)\)%.elementSet().contains($1))
6  (.*)\.contains\(([\S ]*)\)\)%.elementSet().contains($1))
7  \.containsEntry\(([\S ]*), ([\S ]*)\)%.get($1).contains($2)
8  (.*)\.containsEntry\(([\S ]*), ([\S ]*)\)\)%.get($1).contains($2))
9  \.containsEntry\(([\S ]*), ([\S ]*)\)%.get($1).equals($2)
10 (.*)\.containsEntry\(([\S ]*), ([\S ]*)\)\)%.get($1).equals($2))
11 \.containsKey\(([\S ]*)\)%.keySet().contains($1)
12 (.*)\.containsKey\(([\S ]*)\)\)%.keySet().contains($1))
13 \.isEmpty\(\)%.size() == 0
14 !([\S]*)\.isEmpty\(\)%$1.size() != 0
15 \.add\(([\S ]*)\)%.add($1, 1)
16 \.add\(([\S ]*)\)%.setCount($1, 1)
17 ([\S ]*)\.add\(([\S ]*)\)%$1.setCount($2, $1.count($2), $1.count($2) + 1)
18 ([\S ]*)\.add\(([\S ]*), ([\S ]*)\)%for (int _i = 0; _i < $3; _i++) { $1.add($2)↩
       ; }
```

[1] http://docs.oracle.com/javase/7/docs/api/java/util/regex/Pattern.html

```
19 ([a-zA-Z0-9_.()\$"]*)\.add\(([\S ]*), ([0-9]*)\)%int __i = 0;while (__i++ < $3) ↩
       { $1.add($2); }
20 ([\S ]*)\.setCount\(([\S ]*), ([\S ]*), ([\S ]*)\)%$1.remove($2, $3); $1.add($2,↩
        $4)
21 \.remove\(([\S ]*)\)%.remove($1, 1)
22 ([\S]*)\.clear\(\)%$1 = $1.create()
23 ([\S]*)\.clear\(\)%for (int _i = 0; _i < $1.keySet().size(); _i++) { $1.↩
       removeAll($1.keySet().toArray()[_i]); }
24 \.remove\(([a-zA-Z0-9_.()\$"]*)\)%.standardRemove($1)
25 \.remove\(([\S ]*), ([\S ]*)\)%.get($1).remove($2)
26 ([a-zA-Z0-9_.()\$"]*)\.removeAll\(([\S ]*)\)%List __c=$1.get($2); while($1.get(↩
       $2).size()> 0) { $1.remove($2, c.get(0)); }
27 ([a-zA-Z0-9_.()\$"]*)\.removeAll\(([\S ]*)\)%for (Iterator __iterator = $2.↩
       iterator(); __iterator.hasNext();) { $1.remove(__iterator.next()); }
28 \.size\(\)%.keys().size()
29 \.size\(\)%.toArray().length
30 ([a-zA-Z0-9_.()\$"]*)\.putIfAbsent\(([\S ]*), ([\S ]*)\)%if (!$1.containsEntry(↩
       $2, $3)) { $1.put($2, $3); }
31 \.create\(\)%.create(100, 100)
32 \.create\(([0-9]*), ([0-9]*)\)%.create()
33 \.create\(\)%.create(100)
34 \.create\(([0-9]*)\)%.create()
35 ([a-zA-Z0-9_.()\$"]*)\.create\(([\S ]*)\)%$1.create(); $1.putAll($2);
36 \.retainAll\(([\S ]*)\)%.standardRetainAll($1)
37 \.apply\(([\S ]*)\)%.contains($1)
38 ([a-zA-Z0-9_.()\$"]*)\.setCount\(([\S ]*), ([0-9]*)\)%int __i = $1.count($2); if↩
        (1 < __i) { while ($3 < __i--) { $1.remove($2); } } else { while (__i++ < ↩
       1) { $1.add($2); } }
39 \.toArray\(([\S ]*)\)%.toArray()
40 \.of\(\)%.builder()\.build()
41 \.of\(([\S ]*)\)%.builder()\.add($1)\.build()
42 \.copyOf\(([\S ]*)\)%.copyOf($1.toArray())
43 \.copyOf\(([\S ]*)\)%.copyOf($1.iterator())
44 \.newStrongInterner\(\)%.newWeakInterner()
45 \.newWeakInterner\(\)%.newStrongInterner()
46 ([a-zA-Z0-9_.()\$"]*)\.clear\(\)%for (Iterator iterator = $1.keys().iterator(); ↩
       iterator.hasNext();) { $1.removeAll(iterator.next()); }
47 ([a-zA-Z0-9_.()\$"]*)\.replaceValues\(([\S ]*), ([\S ]*)\)%$1.removeAll($2); $1.↩
       putAll($2, $3)
48 ([\S]*) ([\S]*) = ([\S]*)\.newLinkedList\(([\S]*)\)%$1 $2 = $3.newLinkedList(); ↩
       $2.addAll($4)
49 \.newLinkedList\(\)%.newArrayListWithExpectedSize(100)
50 \.newArrayListWithExpectedSize\(([0-9]*)\)%.newArrayList()
51 \.newArrayListWithCapacity\(([0-9]*)\)%.newArrayList()
52 \.newHashMap\(\)%.newHashMapWithExpectedSize(100)
53 \.newHashMap\(([\S ]*)\)%.newHashMap().putAll($1)
54 \.newLinkedHashMap\(([\S ]*)\)%.newLinkedHashMap().putAll($1)
55 \.newTreeMap\(\)%.newTreeMap(com.google.common.collect.Ordering.natural())
56 \.newTreeMap\(([\S ]*)\)%.newTreeMap().putAll($1)
57 \.newLinkedHashSet\(\)%.newLinkedHashSetWithExpectedSize(100)
58 ([\S]*) ([\S]*) = ([\S]*)\.newLinkedHashSet\(([\S]*)\)%$1 $2 = $3.↩
```

```
    newLinkedHashSet(); $2.addAll($4)
59  \.newTreeSet\(\)%.newTreeSet(com.google.common.collect.Ordering.natural())
60  ([\S]*) ([\S]*) = ([\S]*)\.newTreeSet\(([\S]*)\)%$1 $2 = $3.newTreeSet(); $2.↩
        addAll($4)
61  \.newHashSet\(\)%.newHashSetWithExpectedSize(100)
62  \.newHashSetWithExpectedSize\(([?:\S ]*)\)%.newHashSet()
63  ([\S]*) ([\S]*) = ([\S]*)\.newHashSet\(([\S]*)\)%$1 $2 = $3.newHashSet(); $2.↩
        addAll($4)
```

A.2 Code Rewriting Rules for JodaTime

```
1   \.plusDays\((?:\s*)([a-zA-Z0-9_.() \+"]*[^ ])(?:\s*)\)%.plus(org.joda.time.↩
        Period.days($1))
2   \.plusDays\((?:\s*)([a-zA-Z0-9_.() \+"]*[^ ])(?:\s*)\)%.withFieldAdded(org.joda.↩
        time.DurationFieldType.days(), $1)
3   new YearMonthDay\((?:\s*)([a-zA-Z0-9_.()"]*[^ ])(?:\s*)\)%new YearMonthDay($1, ↩
        org.joda.time.chrono.ISOChronology.getInstance())
4   \.parseDateTime\((?:\s*)([a-zA-Z0-9_.() \+:"]*[^ ])(?:\s*)\)%.parseLocalDateTime↩
        ($1)
5   \.forTimeZone\(([a-zA-Z0-9_.\+\-]*\(\))%.forID($1.getID()
6   \.millisOfDay\(\)\.withMinimumValue\(\)%.toDateMidnight().toDateTime()
7   \.millisOfDay\(\)\.withMinimumValue\(\)%.withTimeAtStartOfDay()
8   ([a-zA-Z0-9_.()]*)\.isAfter\((?:\s*)([a-zA-Z0-9_.() \+"]*[^ ])(?:\s*)\)%(!$1.↩
        isBefore($2))
9   ([a-zA-Z0-9_.()]*)\.isBefore\((?:\s*)([a-zA-Z0-9_.() \+"]*[^ ])(?:\s*)\)%(!$1.↩
        isAfter($2))
10  ([a-zA-Z0-9_.()]*)\.isEqual\((?:\s*)([a-zA-Z0-9_.() \+"]*[^ ])(?:\s*)\)%$1.↩
        getMillis() == $2.getMillis()
11  ([a-zA-Z0-9_.()]*)\.isShorterThan\((?:\s*)([a-zA-Z0-9_.() \+"]*[^ ])(?:\s*)\)%$1↩
        .getMillis() < $2.getMillis()
12  ([a-zA-Z0-9_.()]*)\.isLongerThan\((?:\s*)([a-zA-Z0-9_.() \+"]*[^ ])(?:\s*)\)%$1.↩
        getMillis() > $2.getMillis()
13  ([a-zA-Z0-9_.()]*)\.isAfter\((?:\s*)([a-zA-Z0-9_.() \+"]*[^ ])(?:\s*)\)%$1.↩
        getMillis() > $2
14  ([a-zA-Z0-9_.()]*)\.isBefore\((?:\s*)([a-zA-Z0-9_.() \+"]*[^ ])(?:\s*)\)%$1.↩
        getMillis() < $2
15  ([a-zA-Z0-9_.()]*)\.isEqual\((?:\s*)([a-zA-Z0-9_.() \+"]*[^ ])(?:\s*)\)%$1.↩
        getMillis() == $2
16  new DateTime\(\)%new DateTime(org.joda.time.chrono.ISOChronology.getInstance())
17  new DateTime\(([a-zA-Z0-9_.()]*)\)%new DateTime($1, org.joda.time.chrono.↩
        ISOChronology.getInstance())
18  new DateTime\((?: *)([a-zA-Z0-9_.()]*)(?: *),(?: *)([a-zA-Z0-9_.()]*)(?: *),(?: ↩
        *)([a-zA-Z0-9_.()]*)(?: *),(?: *)([a-zA-Z0-9_.()]*)(?: *),(?: *)([a-zA-Z0-9_↩
        .()]*)(?: *)\)%new DateTime($1, $2, $3, $4, $5, org.joda.time.chrono.↩
        ISOChronology.getInstance())
19  new DateTime\((?: *)([a-zA-Z0-9_.()]*)(?: *),(?: *)([a-zA-Z0-9_.()]*)(?: *),(?: ↩
        *)([a-zA-Z0-9_.()]*)(?: *),(?: *)([a-zA-Z0-9_.()]*)(?: *),(?: *)([a-zA-Z0-9_↩
        .()]*)(?: *),(?: *)([a-zA-Z0-9_.()]*)(?: *)\)%new DateTime($1, $2, $3, $4, ↩
        $5, $6, org.joda.time.chrono.ISOChronology.getInstance())
20  new DateTime\((?: *)([a-zA-Z0-9_.()]*)(?: *),(?: *)([a-zA-Z0-9_.()]*)(?: *),(?: ↩
```

```
    *)([a-zA-Z0-9_.()]*)(?: *),(?: *)([a-zA-Z0-9_.()]*)(?: *),(?: *)([a-zA-Z0-9_↩
    .()]*)(?: *),(?: *)([a-zA-Z0-9_.()]*)(?: *),(?: *)([a-zA-Z0-9_.()]*)(?: *)\)↩
    %new DateTime($1, $2, $3, $4, $5, $6, $7, org.joda.time.chrono.ISOChronology↩
    .getInstance())
([a-zA-Z0-9_.()]*)\.minus\((?: *)([a-zA-Z0-9_.() \+"]*)(?: *)\)%$1.minus($2.↩
    getMillis())
([a-zA-Z0-9_.()]*)\.minus\((?: *)([a-zA-Z0-9_.() \+"]*)(?: *)\)%$1.minusMillis(↩
    $2.getMillis())
([a-zA-Z0-9_.()]*)\.plus\((?: *)([a-zA-Z0-9_.() \+"]*)(?: *)\)%$1.plus($2.↩
    getMillis())
([a-zA-Z0-9_.()]*).plus\((?: *)([a-zA-Z0-9_.() \+"]*)(?: *)\)%$1.plusMillis($2.↩
    getMillis())
\.now\(\)%.now(org.joda.time.chrono.ISOChronology.getInstance())
\.toDateTime\(\)%.toDateTime(org.joda.time.chrono.ISOChronology.getInstance())
\.toDateTimeISO\(\)%.toDateTime(org.joda.time.chrono.ISOChronology.getInstance()↩
    )
\.getCenturyOfEra\(\)%.centuryOfEra().get()
\.getDayOfMonth\(\)%.dayOfMonth().get()
\.getDayOfWeek\(\)%.dayOfWeek().get()
\.getDayOfYear\(\)%.dayOfYear().get()
\.getHourOfDay\(\)%.hourOfDay().get()
\.getMillisOfDay\(\)%.millisOfDay().get()
\.getMillisOfSecond\(\)%.millisOfSecond().get()
\.getMinuteOfDay\(\)%.minuteOfDay().get()
\.getMinuteOfHour\(\)%.minuteOfHour().get()
\.getMonthOfYear\(\)%.monthOfYear().get()
\.getSecondOfDay\(\)%.secondOfDay().get()
\.getSecondOfMinute\(\)%.secondOfMinute().get()
\.getWeekOfWeekYear\(\)%.weekOfWeekYear().get()
\.getYear\(\)%.year().get()
\.getYearOfCentury\(\)%.yearOfCentury().get()
\.getYearOfEra\(\)%.yearOfEra().get()
\.getYearOfEra\(\)%.getYear()
\.getYear\(\)%.getWeekyear()
\.centuryOfEra\(\)\.get\(\)%.getCenturyOfEra()
\.dayOfMonth\(\)\.get\(\)%.getDayOfMonth()
\.dayOfWeek\(\)\.get\(\)%.getDayOfWeek()
\.era\(\)\.get\(\)%.getEra()
\.hourOfDay\(\)\.get\(\)%.getHourOfDay()
\.millisOfDay\(\)\.get\(\)%.getMillisOfDay()
\.millisOfSecond\(\)\.get\(\)%.getMillisOfSecond()
\.minuteOfDay\(\)\.get\(\)%.getMinuteOfDay()
\.minuteOfHour\(\)\.get\(\)%.getMinuteOfHour()
\.monthOfYear\(\)\.get\(\)%.getMonthOfYear()
\.secondOfDay\(\)\.get\(\)%.getSecondOfDay()
\.secondOfMinute\(\)\.get\(\)%.getSecondOfMinute()
\.weekOfWeekYear\(\)\.get\(\)%.getWeekOfWeekYear()
\.year\(\)\.get\(\)%.getYear()
\.yearOfCentury\(\)\.get\(\)%.getYearOfCentury()
\.getYearOfEra\(\)\.get\(\)%.yearOfEra()
\.getYear\(\)\.get\(\)%.getYearOfEra()
```

```
63 \.getWeekyear\(\)\.get\(\)%.getYear()
64 \.getYear\(\)%.minusYears(1).getYear() + 1
65 \.getDayOfMonth\(\)%.minusDays(1).getDayOfMonth() + 1
66 \.getHourOfDay\(\)%.minusHours(1).getHours() + 1
67 \.getMillisOfSecond\(\)%.minusMillis(1).getMillisOfSecond() + 1
68 \.getMinuteOfHour\(\)%.minusMinutes(1).getMinuteOfHour() + 1
69 \.getMonthOfYear\(\)%.minusMonths(1).getMonthOfYear() + 1
70 \.getSecondOfMinute\(\)%.minusSeconds(1).getSecondOfMinute() + 1
71 \.getWeekOfWeekYear\(\)%.minusWeeks(1).getWeekOfWeekYear() + 1
72 \.getYear\(\)%.plusYears(1).getYear() - 1
73 \.getDayOfMonth\(\)%.plusDays(1).getDayOfMonth() - 1
74 \.getHourOfDay\(\)%.plusHours(1).getHours() - 1
75 \.getMillisOfSecond\(\)%.plusMillis(1).getMillisOfSecond() - 1
76 \.getMinuteOfHour\(\)%.plusMinutes(1).getMinuteOfHour() - 1
77 \.getMonthOfYear\(\)%.plusMonths(1).getMonthOfYear() - 1
78 \.getSecondOfMinute\(\)%.plusSeconds(1).getSecondOfMinute() - 1
79 \.getWeekOfWeekYear\(\)%.plusWeeks(1).getWeekOfWeekYear() - 1
80 Period\((?: *)([ a-zA-Z0-9_.()\+\-]*)(?: *),(?: *)([ a-zA-Z0-9_.()\+\-]*)(?: *)↩
       ,(?: *)([ a-zA-Z0-9_.()\+\-]*)(?: *),(?: *)([ a-zA-Z0-9_.()\+\-]*)(?: *)\)%↩
       Period().plusHours($1).plusMinutes($2).plusSeconds($3).plusMillis($4)
81 Period\((?: *)([ a-zA-Z0-9_.()\+\-]*)(?: *),(?: *)([ a-zA-Z0-9_.()\+\-]*)(?: *)↩
       ,(?: *)([ a-zA-Z0-9_.()\+\-]*)(?: *),(?: *)([ a-zA-Z0-9_.()\+\-]*)(?: *),(?:↩
        *)([ a-zA-Z0-9_.()\+\-]*)(?: *),(?: *)([ a-zA-Z0-9_.()\+\-]*)(?: *),(?: *)↩
       ([ a-zA-Z0-9_.()\+\-]*)(?: *),(?: *)([ a-zA-Z0-9_.()\+\-]*)(?: *)\)%Period()↩
       .plusYears($1).plusMonths($2).plusWeeks($3).plusDays($4).plusHours($5).↩
       plusMinutes($6).plusSeconds($7).plusMillis($8)
82 Period\((?: *)([ a-zA-Z0-9_.()\+\-]*)(?: *),(?: *)([ a-zA-Z0-9_.()\+\-]*)(?: *)↩
       ,(?: *)([ a-zA-Z0-9_.()\+\-]*)(?: *),(?: *)([ a-zA-Z0-9_.()\+\-]*)(?: *),(?:↩
        *)([ a-zA-Z0-9_.()\+\-]*)(?: *),(?: *)([ a-zA-Z0-9_.()\+\-]*)(?: *),(?: *)↩
       ([ a-zA-Z0-9_.()\+\-]*)(?: *),(?: *)([ a-zA-Z0-9_.()\+\-]*)(?: *),(?: *)([ a↩
       -zA-Z0-9_.()]*)(?: *)\)%Period().plusYears($1).plusMonths($2).plusWeeks($3).↩
       plusDays($4).plusHours($5).plusMinutes($6).plusSeconds($7).plusMillis($8).↩
       withPeriodType($9)
83 Period\((?: *)([ a-zA-Z0-9_.()\+\-]*)(?: *),(?: *)([ a-zA-Z0-9_.()\+\-]*)(?: *)↩
       ,(?: *)([ a-zA-Z0-9_.()\+\-]*)(?: *),(?: *)([ a-zA-Z0-9_.()\+\-]*)(?: *)\)%↩
       Period().withHours($1).withMinutes($2).withSeconds($3).withMillis($4)
84 Period\((?: *)([ a-zA-Z0-9_.()\+\-]*)(?: *),(?: *)([ a-zA-Z0-9_.()\+\-]*)(?: *)↩
       ,(?: *)([ a-zA-Z0-9_.()\+\-]*)(?: *),(?: *)([ a-zA-Z0-9_.()\+\-]*)(?: *),(?:↩
        *)([ a-zA-Z0-9_.()\+\-]*)(?: *),(?: *)([ a-zA-Z0-9_.()\+\-]*)(?: *),(?: *)↩
       ([ a-zA-Z0-9_.()\+\-]*)(?: *),(?: *)([ a-zA-Z0-9_.()\+\-]*)(?: *)\)%Period()↩
       .withYears($1).withMonths($2).withWeeks($3).withDays($4).withHours($5).↩
       withMinutes($6).withSeconds($7).withMillis($8)
85 Period\((?: *)([ a-zA-Z0-9_.()\+\-]*)(?: *),(?: *)([ a-zA-Z0-9_.()\+\-]*)(?: *)↩
       ,(?: *)([ a-zA-Z0-9_.()\+\-]*)(?: *),(?: *)([ a-zA-Z0-9_.()\+\-]*)(?: *),(?:↩
        *)([ a-zA-Z0-9_.()\+\-]*)(?: *),(?: *)([ a-zA-Z0-9_.()\+\-]*)(?: *),(?: *)↩
       ([ a-zA-Z0-9_.()\+\-]*)(?: *),(?: *)([ a-zA-Z0-9_.()\+\-]*)(?: *),(?: *)([ a↩
       -zA-Z0-9_.()]*)(?: *)\)%Period().withYears($1).withMonths($2).withWeeks($3).↩
       withDays($4).withHours($5).withMinutes($6).withSeconds($7).withMillis($8).↩
       withPeriodType($9)
86 Period\((?: *)([ a-zA-Z0-9_.()\+\-]*)(?: *),(?: *)([ a-zA-Z0-9_.()\+\-]*)(?: *)↩
       ,(?: *)([ a-zA-Z0-9_.()\+\-]*)(?: *),(?: *)([ a-zA-Z0-9_.()\+\-]*)(?: *)\)%↩
```

```
    Period().withField(org.joda.time.DurationFieldType.hours(), $1).withField(↩
    org.joda.time.DurationFieldType.minutes(), $2).withField(org.joda.time.↩
    DurationFieldType.seconds(), $3).withField(org.joda.time.DurationFieldType.↩
    millis(), $4)
87  Period\((?: *)([ a-zA-Z0-9_.()\+\-]*)(?: *),(?: *)([ a-zA-Z0-9_.()\+\-]*)(?: *)↩
    ,(?: *)([ a-zA-Z0-9_.()\+\-]*)(?: *),(?: *)([ a-zA-Z0-9_.()\+\-]*)(?: *),(?:↩
     *)([ a-zA-Z0-9_.()\+\-]*)(?: *),(?: *)([ a-zA-Z0-9_.()\+\-]*)(?: *),(?: *)↩
    ([ a-zA-Z0-9_.()\+\-]*)(?: *),(?: *)([ a-zA-Z0-9_.()\+\-]*)(?: *)\)%Period()↩
    .withField(org.joda.time.DurationFieldType.years(), $1).withField(org.joda.↩
    time.DurationFieldType.months(), $2).withField(org.joda.time.↩
    DurationFieldType.weeks(), $3).withField(org.joda.time.DurationFieldType.↩
    days(), $4).withField(org.joda.time.DurationFieldType.hours(), $5).withField↩
    (org.joda.time.DurationFieldType.minutes(), $6).withField(org.joda.time.↩
    DurationFieldType.seconds(), $7).withField(org.joda.time.DurationFieldType.↩
    millis(), $8)
88  Period\((?: *)([ a-zA-Z0-9_.()\+\-]*)(?: *),(?: *)([ a-zA-Z0-9_.()\+\-]*)(?: *)↩
    ,(?: *)([ a-zA-Z0-9_.()\+\-]*)(?: *),(?: *)([ a-zA-Z0-9_.()\+\-]*)(?: *),(?:↩
     *)([ a-zA-Z0-9_.()\+\-]*)(?: *),(?: *)([ a-zA-Z0-9_.()\+\-]*)(?: *),(?: *)↩
    ([ a-zA-Z0-9_.()\+\-]*)(?: *),(?: *)([ a-zA-Z0-9_.()\+\-]*)(?: *),(?: *)([ a↩
    -zA-Z0-9_.()]*)(?: *)\)%Period().withField(org.joda.time.DurationFieldType.↩
    years(), $1).withField(org.joda.time.DurationFieldType.months(), $2).↩
    withField(org.joda.time.DurationFieldType.weeks(), $3).withField(org.joda.↩
    time.DurationFieldType.days(), $4).withField(org.joda.time.DurationFieldType↩
    .hours(), $5).withField(org.joda.time.DurationFieldType.minutes(), $6).↩
    withField(org.joda.time.DurationFieldType.seconds(), $7).withField(org.joda.↩
    time.DurationFieldType.millis(), $8).withPeriodType($9)
89  Period\((?: *)([ a-zA-Z0-9_.()\+\-]*)(?: *),(?: *)([ a-zA-Z0-9_.()\+\-]*)(?: *)↩
    ,(?: *)([ a-zA-Z0-9_.()\+\-]*)(?: *),(?: *)([ a-zA-Z0-9_.()\+\-]*)(?: *)\)%↩
    Period().withFieldAdded(org.joda.time.DurationFieldType.hours(), $1).↩
    withFieldAdded(org.joda.time.DurationFieldType.minutes(), $2).withFieldAdded↩
    (org.joda.time.DurationFieldType.seconds(), $3).withFieldAdded(org.joda.time↩
    .DurationFieldType.millis(), $4)
90  Period\((?: *)([ a-zA-Z0-9_.()\+\-]*)(?: *),(?: *)([ a-zA-Z0-9_.()\+\-]*)(?: *)↩
    ,(?: *)([ a-zA-Z0-9_.()\+\-]*)(?: *),(?: *)([ a-zA-Z0-9_.()\+\-]*)(?: *),(?:↩
     *)([ a-zA-Z0-9_.()\+\-]*)(?: *),(?: *)([ a-zA-Z0-9_.()\+\-]*)(?: *),(?: *)↩
    ([ a-zA-Z0-9_.()\+\-]*)(?: *),(?: *)([ a-zA-Z0-9_.()\+\-]*)(?: *)\)%Period()↩
    .withFieldAdded(org.joda.time.DurationFieldType.years(), $1).withFieldAdded(↩
    org.joda.time.DurationFieldType.months(), $2).withFieldAdded(org.joda.time.↩
    DurationFieldType.weeks(), $3).withFieldAdded(org.joda.time.↩
    DurationFieldType.days(), $4).withFieldAdded(org.joda.time.DurationFieldType↩
    .hours(), $5).withFieldAdded(org.joda.time.DurationFieldType.minutes(), $6).↩
    withFieldAdded(org.joda.time.DurationFieldType.seconds(), $7).withFieldAdded↩
    (org.joda.time.DurationFieldType.millis(), $8)
91  Period\((?: *)([ a-zA-Z0-9_.()\+\-]*)(?: *),(?: *)([ a-zA-Z0-9_.()\+\-]*)(?: *)↩
    ,(?: *)([ a-zA-Z0-9_.()\+\-]*)(?: *),(?: *)([ a-zA-Z0-9_.()\+\-]*)(?: *),(?:↩
     *)([ a-zA-Z0-9_.()\+\-]*)(?: *),(?: *)([ a-zA-Z0-9_.()\+\-]*)(?: *),(?: *)↩
    ([ a-zA-Z0-9_.()\+\-]*)(?: *),(?: *)([ a-zA-Z0-9_.()\+\-]*)(?: *),(?: *)([ a↩
    -zA-Z0-9_.()]*)(?: *)\)%Period().withFieldAdded(org.joda.time.↩
    DurationFieldType.years(), $1).withFieldAdded(org.joda.time.↩
    DurationFieldType.months(), $2).withFieldAdded(org.joda.time.↩
    DurationFieldType.weeks(), $3).withFieldAdded(org.joda.time.↩
```

```
    DurationFieldType.days(), $4).withFieldAdded(org.joda.time.DurationFieldType↩
    .hours(), $5).withFieldAdded(org.joda.time.DurationFieldType.minutes(), $6).↩
    withFieldAdded(org.joda.time.DurationFieldType.seconds(), $7).withFieldAdded↩
    (org.joda.time.DurationFieldType.millis(), $8).withPeriodType($9)
92  MutablePeriod(?: *)([a-zA-Z0-9_]*)(?: *)=(?: *)new(?: *)MutablePeriod\((?: *)([ ↩
    a-zA-Z0-9_.()\+\-]*)(?: *),(?: *)([ a-zA-Z0-9_.()\+\-]*)(?: *),(?: *)([ a-zA↩
    -Z0-9_.()\+\-]*)(?: *),(?: *)([ a-zA-Z0-9_.()\+\-]*)(?: *)\)(?: *);%↩
    MutablePeriod $1 = new MutablePeriod(); $1.add(org.joda.time.↩
    DurationFieldType.hours(), $2); $1.add(org.joda.time.DurationFieldType.↩
    minutes(), $3); $1.add(org.joda.time.DurationFieldType.seconds(), $4); $1.↩
    add(org.joda.time.DurationFieldType.millis(), $5);
93  MutablePeriod(?: *)([a-zA-Z0-9_]*)(?: *)=(?: *)new(?: *)MutablePeriod\((?: *)([ ↩
    a-zA-Z0-9_.()\+\-]*)(?: *),(?: *)([ a-zA-Z0-9_.()\+\-]*)(?: *),(?: *)([ a-zA↩
    -Z0-9_.()\+\-]*)(?: *),(?: *)([ a-zA-Z0-9_.()\+\-]*)(?: *)\)(?: *);%↩
    MutablePeriod $1 = new MutablePeriod(); $1.addHours($2); $1.addMinutes($3); ↩
    $1.addSeconds($4); $1.addMillis($5);
94  MutablePeriod(?: *)([a-zA-Z0-9_]*)(?: *)=(?: *)new(?: *)MutablePeriod\((?: *)([ ↩
    a-zA-Z0-9_.()\+\-]*)(?: *),(?: *)([ a-zA-Z0-9_.()\+\-]*)(?: *),(?: *)([ a-zA↩
    -Z0-9_.()\+\-]*)(?: *),(?: *)([ a-zA-Z0-9_.()\+\-]*)(?: *)\)(?: *);%↩
    MutablePeriod $1 = new MutablePeriod(); $1.setHours($2); $1.setMinutes($3); ↩
    $1.setSeconds($4); $1.setMillis($5);
95  MutablePeriod(?: *)([a-zA-Z0-9_]*)(?: *)=(?: *)new(?: *)MutablePeriod\((?: *)([ ↩
    a-zA-Z0-9_.()\+\-]*)(?: *),(?: *)([ a-zA-Z0-9_.()\+\-]*)(?: *),(?: *)([ a-zA↩
    -Z0-9_.()\+\-]*)(?: *),(?: *)([ a-zA-Z0-9_.()\+\-]*)(?: *),(?: *)([ a-zA-Z0↩
    -9_.()\+\-]*)(?: *),(?: *)([ a-zA-Z0-9_.()\+\-]*)(?: *),(?: *)([ a-zA-Z0-9_↩
    .()\+\-]*)(?: *),(?: *)([ a-zA-Z0-9_.()\+\-]*)(?: *)\)(?: *);%MutablePeriod ↩
    $1 = new MutablePeriod(); $1.add(org.joda.time.DurationFieldType.years(), $2↩
    ); $1.add(org.joda.time.DurationFieldType.months(), $3); $1.add(org.joda.↩
    time.DurationFieldType.weeks(), $4); $1.add(org.joda.time.DurationFieldType.↩
    days(), $5); $1.add(org.joda.time.DurationFieldType.hours(), $6); $1.add(org↩
    .joda.time.DurationFieldType.minutes(), $7); $1.add(org.joda.time.↩
    DurationFieldType.seconds(), $8); $1.add(org.joda.time.DurationFieldType.↩
    millis(), $9);
96  MutablePeriod(?: *)([a-zA-Z0-9_]*)(?: *)=(?: *)new(?: *)MutablePeriod\((?: *)([ ↩
    a-zA-Z0-9_.()\+\-]*)(?: *),(?: *)([ a-zA-Z0-9_.()\+\-]*)(?: *),(?: *)([ a-zA↩
    -Z0-9_.()\+\-]*)(?: *),(?: *)([ a-zA-Z0-9_.()\+\-]*)(?: *),(?: *)([ a-zA-Z0↩
    -9_.()\+\-]*)(?: *),(?: *)([ a-zA-Z0-9_.()\+\-]*)(?: *),(?: *)([ a-zA-Z0-9_↩
    .()\+\-]*)(?: *),(?: *)([ a-zA-Z0-9_.()\+\-]*)(?: *)\)(?: *);%MutablePeriod ↩
    $1 = new MutablePeriod(); $1.addYears($2); $1.addMonths($3); $1.addWeeks($4)↩
    ; $1.addDays($5); $1.addHours($6); $1.addMinutes($7); $1.addSeconds($8); $1.↩
    addMillis($9);
97  MutablePeriod(?: *)([a-zA-Z0-9_]*)(?: *)=(?: *)new(?: *)MutablePeriod\((?: *)([ ↩
    a-zA-Z0-9_.()\+\-]*)(?: *),(?: *)([ a-zA-Z0-9_.()\+\-]*)(?: *),(?: *)([ a-zA↩
    -Z0-9_.()\+\-]*)(?: *),(?: *)([ a-zA-Z0-9_.()\+\-]*)(?: *),(?: *)([ a-zA-Z0↩
    -9_.()\+\-]*)(?: *),(?: *)([ a-zA-Z0-9_.()\+\-]*)(?: *),(?: *)([ a-zA-Z0-9_↩
    .()\+\-]*)(?: *),(?: *)([ a-zA-Z0-9_.()\+\-]*)(?: *)\)(?: *);%MutablePeriod ↩
    $1 = new MutablePeriod(); $1.setYears($2); $1.setMonths($3); $1.setWeeks($4)↩
    ; $1.setDays($5); $1.setHours($6); $1.setMinutes($7); $1.setSeconds($8); $1.↩
    setMillis($9);
98  MutablePeriod(?: *)([a-zA-Z0-9_]*)(?: *)=(?: *)new(?: *)MutablePeriod\((?: *)([ ↩
    a-zA-Z0-9_.()\+\-]*)(?: *)\)(?: *);%MutablePeriod $1 = new MutablePeriod(); ↩
```

```
    $1.add($2);
99  MutablePeriod(?: *)([a-zA-Z0-9_]*)(?: *)=(?: *)new(?: *)MutablePeriod\((?: *)([ ↩
    a-zA-Z0-9_.()\+\-]*)(?: *)\)(?: *);%MutablePeriod $1 = new MutablePeriod(); ↩
    $1.setPeriod($2);
100 MutablePeriod(?: *)([a-zA-Z0-9_]*)(?: *)=(?: *)new(?: *)MutablePeriod\((?: *)([ ↩
    a-zA-Z0-9_.()\+\-]*)(?: *),(?: *)([ a-zA-Z0-9_.()\+\-]*)(?: *)\)(?: *);%↩
    MutablePeriod $1 = new MutablePeriod(); $1.setPeriod($2, $3);
```

Bibliography

[ABL05] J.H. Andrews, L.C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, pages 402–411, 2005.

[AGGM04] Saurabh Agarwal, Rahul Garg, Meeta S. Gupta, and Jose E. Moreira. Adaptive incremental checkpointing for massively parallel systems. In *Proceedings of the 18th Annual International Conference on Supercomputing*, ICS '04, pages 277–286, 2004.

[AK88] Paul E. Ammann and John C. Knight. Data diversity: An approach to software fault tolerance. *IEEE Transactions on Computers*, 37(4):418–425, 1988.

[ATLM+06] Ali-Reza Adl-Tabatabai, Brian T. Lewis, Vijay Menon, Brian R. Murphy, Bratin Saha, and Tatiana Shpeisman. Compiler and runtime support for efficient software transactional memory. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, pages 26–37, 2006.

[Avi85] Algirdas Antanas Avizienis. The N-version approach to fault-tolerant software. *IEEE Transactions on Software Engineering*, 11(12):1491–1501, 1985.

[AY08] Andrea Arcuri and Xin Yao. A novel co-evolutionary approach to automatic software bug fixing. In *Proceedings of the IEEE Congress on Evolutionary Computation*, CEC '08, pages 162–168, 2008.

[BDK+08] Evgueni Brevnov, Yuri Dolgov, Boris Kuznetsov, Dmitry Yershov, Vyacheslav Shakin, Dong-Yuan Chen, Vijay Menon, and Suresh Srinivas. Practical experiences with java software transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '08, pages 287–288, 2008.

[BG07] Luciano Baresi and Sam Guinea. Dynamo and self-healing bpel compositions. In *Companion to the Proceedings of the 29th International Con-*

ference on Software Engineering, ICSE COMPANION '07, pages 69–70, 2007.

[BGH+07] David. Breitgand, Mark Goldstein, Ealan Henis, Onn Shehory, and Yaron Weinsberg. Panacea towards a self-healing development framework. In *Proceedings of the 10th International Symposium on Integrated Network Management*, IM '07, pages 169–178, 2007.

[BGP07] Luciano Baresi, Sam Guinea, and Liliana Pasquale. Self-healing bpel processes with dynamo and the jboss rule engine. In *Proceedings of the International Workshop on Engineering of Software Services for Pervasive Environments*, ESSPE '07, pages 11–20, 2007.

[BMP09] Anton Babenko, Leonardo Mariani, and Fabrizio Pastore. Ava: Automated interpretation of dynamically detected anomalies. In *Proceedings of the 18th International Symposium on Software Testing and Analysis*, ISSTA '09, pages 237–248, 2009.

[BP65] Richard E. Barlow and Frank Proschan. *Mathematical Theory of Reliability*. John Wiley & Sons, Inc., New York, 1965.

[CBM+08] Calin Cascaval, Colin Blundell, Maged Michael, Harold W. Cain, Peng Wu, Stefanie Chiras, and Siddhartha Chatterjee. Software transactional memory: Why is it only a research toy? *Queue - The Concurrency Problem*, 6:46–58, 2008.

[CCF+02] George Candea, James Cutler, Armando Fox, Rushabh Doshi, Priyank Garg, and Rakesh Gowda. Reducing recovery time in a small recursively restartable system. In *Proceedings of the 32th International Conference on Dependable Systems and Networks*, DSN '02, pages 605–614, 2002.

[CDP+09] Antonio Carzaniga, Giovanni Denaro, Mauro Pezzè, Jacky Estublier, and Alexander L. Wolf. Toward deeply adaptive societies of digital systems. In *Companion to the Proceedings of the 31st International Conference on Software Engineering*, ICSE COMPANION '09, pages 331–334, 2009.

[CGP08a] Antonio Carzaniga, Alessandra Gorla, and Mauro Pezzè. Healing web applications through automatic workarounds. *International Journal on Software Tools for Technology Transfer*, 10(6):493–502, 2008.

[CGP08b] Antonio Carzaniga, Alessandra Gorla, and Mauro Pezzè. Self-healing by means of automatic workarounds. In *Proceedings of the International Workshop on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS '08, pages 17–24, 2008.

[CGPP10a] Antonio Carzaniga, Alessandra Gorla, Nicolò Perino, and Mauro Pezzè. Automatic workarounds for web applications. In *Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '10, pages 237–246, 2010.

[CGPP10b] Antonio Carzaniga, Alessandra Gorla, Nicolò Perino, and Mauro Pezzè. Raw: Runtime automatic workarounds. In *Proceedings of the 32nd International Conference on Software Engineering*, ICSE '10, pages 321–322, 2010.

[CKF+04] George Candea, Shinichi Kawamoto, Yuichi Fujiki, Greg Friedman, and Armando Fox. Microreboot — a technique for cheap recovery. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation*, OSDI '04, page 3, 2004.

[CM11] Bruno Cabral and Paulo Marques. A transactional model for automatic exception handling. *Computer Languages, Systems and Structures*, 37:43–61, 2011.

[CMP09] Herve Chang, Leonardo Mariani, and Mauro Pezzè. In-field healing of integration problems with COTS components. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 166–176, 2009.

[CPW72] John R. Connet, Edward J. Pasternak, and Bruce D. Wagner. Software defenses in real-time control systems. In *Proceedings of the 2nd International Symposium on Fault-Tolerant Computing*, FTCS '72, pages 94–99, 1972.

[CR72] K. Mani Chandy and V. Chittoor Ramamoorthy. Rollback and recovery strategies for computer programs. *IEEE Transactions on Computers*, 21:546–556, 1972.

[Cri82] F. Cristian. Exception handling and software fault tolerance. *IEEE Transactions on Computers*, C-31(6):531–540, 1982.

[CRL03] Miguel Castro, Rodrigo Rodrigues, and Barbara Liskov. Base: Using abstraction to improve fault tolerance. *ACM Transactions on Computer Systems*, 21(3):236–269, 2003.

[CRMG12] Iván Cores, Gabriel Rodriguez, Maria J. Martin, and Patricia González. Reducing application-level checkpoint file sizes: Towards scalable fault tolerance solutions. In *Proceedings of the 10th International Symposium on Parallel and Distributed Processing with Applications*, ISPA '12, pages 371–378, 2012.

[CRS06] João Cachopo and António Rito-Silva. Versioned boxes as the basis for memory transactions. *Science of Computer Programming - Special Issue: Synchronization and Concurrency in Object-Oriented Languages*, 63:172–185, 2006.

[CSSS09] Shao Changheng, Fengjing Shao, Xiaoning Song, and Rencheng Sun. A dynamic checkpointing and rollback recovery solution based on task switching. In *Proceedings of the International Symposium on Intelligent Information Systems and Applications*, IISA'09, pages 354–358, 2009.

[dCBGU11] Guido de Caso, Víctor Braberman, Diego Garbervetsky, and Sebastián Uchitel. Program abstractions for behaviour validation. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 381–390, 2011.

[DD08] Brian Demsky and Alokika Dash. Bristlecone: A language for robust software systems. In *Proceedings of the 22nd European Conference on Object-Oriented Programming*, ECOOP '08, pages 490–515, 2008.

[DDZS09] Laura Dietz, Valentin Dallmeier, Andreas Zeller, and Tobias Scheffer. Localizing bugs in program executions with graphical model. In *Proceedings of the 24th Annual Conference on Neural Information Processing Systems*, NIPS '09, pages 468–476, 2009.

[DKT02] Tadashi Dohi, Naoto Kaio, and Kishor S. Trivedi. Availability models with age-dependent checkpointing. In *Proceedings of the 21st Symposium on Reliable Distributed Systems*, SRDS '02, page 130, 2002.

[Dob06] Glen Dobson. Using WS-BPEL to implement software fault tolerance for web services. In *Proceedings of the 32nd Euromicro Conference on Software Engineering and Advanced Applications*, Euromicro '06, pages 126–133, 2006.

[DOK02] T. Dohi, H. Okamura, and N. Kaio. Optimal age-dependent checkpoint strategy with retry of rollback recovery. In *Proceedings of the 2nd International Workshop on Autonomous Decentralized System*, ADS '02, pages 113–118, 2002.

[DPT13] Giovanni Denaro, Mauro Pezzè, and Davide Tosi. Test-and-adapt: An approach for improving service interchangeability. *ACM Transactions on Software Engineering and Methodology*, 22(4):28:1–28:43, 2013.

[DW10] Vidroha Debroy and W. Eric Wong. Using mutation to automatically suggest fixes for faulty programs. In *Proceedings of the 3rd International Conference on Software Testing, Verification and Validation*, ICST '10, pages 65–74, 2010.

[DZM09] Valentin Dallmeier, Andreas Zeller, and Bertrand Meyer. Generating fixes from object behavior anomalies. In *Proceedings of the 24th International Conference on Automated Software Engineering*, ASE '09, pages 550–554, 2009.

[DZM10] Brian Demsky, Jin Zhou, and William Montaz. Recovery tasks: An automated approach to failure recovery. In *Proceedings of the 1st International Conference on Runtime Verification*, RV '10, pages 229–244, 2010.

[EAWJ02] Mootaz Elnozahy, Lorenzo Alvisi, Yi-min Wang, and David B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys*, 34(3):375–408, 2002.

[EF05] Michael Engel and Bernd Freisleben. Supporting autonomic computing functionality via dynamic operating system kernel aspects. In *Proceedings of the 4th International Conference on Aspect-Oriented Software Development*, AOSD '05, pages 51–62, 2005.

[EJZ92] E.N. Elnozahy, D.B. Johnson, and W. Zwaenepoel. The performance of consistent checkpointing. In *Proceedings of the 11st Symposium on Reliable Distributed Systems*, SRDS '11, pages 39–47, 1992.

[ELSC13] Ifeanyi P. Egwutuoha, David Levy, Bran Selic, and Shiping Chen. A survey of fault tolerance mechanisms and checkpoint/restart implementations for high performance computing systems. *The Journal of Supercomputing*, 65(3):1302–1326, 2013.

[EPG+07] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1–3):35–45, 2007.

[Fen05a] Thomas Huining Feng. Extending java with checkpointing. Ptolemy Group, EECS, UC Berkeley, 2005.

[Fen05b] Thomas Huining Feng. Incremental checkpointing based on java source code refactoring. Technical report, Ptolemy Group, CHESS (Center for Hybrid and Embedded Software Systems) EECS, UC Berkeley, 2005.

[FSS02] Robert T. Futrell, Donald F. Shafer, and Linda I. Shafer. *Quality Software Project Management*. Prentice Hall, 2002.

[Gel76] Erol Gelenbe. A model of roll-back recovery with multiple checkpoints. In *Proceedings of the 2nd international conference on Software engineering*, ICSE '76, pages 251–255, Los Alamitos, CA, USA, 1976. IEEE Computer Society Press.

[Gel79] Erol Gelenbe. On the optimum checkpoint interval. *Journal of the ACM*, 26(2):259–270, 1979.

[GHKT96] Sachin Garg, Yennun Huang, Chandra Kintala, and Kishor S. Trivedi. Minimizing completion time of a program by checkpointing and rejuvenation. *SIGMETRICS Performance Evaluation Review*, 24(1):252–261, 1996.

[GLVT06] M. Grottke, Lei Li, K. Vaidyanathan, and K.S. Trivedi. Analysis of software aging in a web server. *IEEE Transactions on Reliability*, 55(3):411–420, 2006.

[GMM07] Carlo Ghezzi, Andrea Mocci, and Mattia Monga. Efficient recovery of algebraic specifications for stateful components. In *Proceedings of the 9th International Workshop on Principles of Software Evolution*, IWPSE '07, pages 98–105, 2007.

[Goo75] John B. Goodenough. Exception handling: Issues and a proposed notation. *Communications of the ACM*, 18:683–696, 1975.

[GPSS04] Ilir Gashi, Peter Popov, Vladimir Stankovic, and Lorenzo Strigini. On designing dependable services with diverse off-the-shelf sql servers. In Rogério de Lemos, Cristina Gacek, and Alexander Romanovsky, editors, *Architecting Dependable Systems II*, volume 3069 of *Lecture Notes in Computer Science*, pages 191–214. Springer Berlin Heidelberg, 2004.

[GR92] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., 1992.

[Gra81] Jim Gray. The transaction concept: Virtues and limitations. In *Proceedings of the 7th International Conference on Very Large Databases*, VLDB '81, pages 144–154, 1981.

[GT07] M. Grottke and K.S. Trivedi. Fighting bugs: Remove, retry, replicate, and rejuvenate. *Computer*, 40(2):107–109, 2007.

[Gui05] Sam Guinea. Self-healing web service compositions. In *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, pages 655–655, 2005.

[Guo11] Philip J. Guo. Sloppy python: Using dynamic analysis to automatically add error tolerance to ad-hoc data processing scripts. In *Proceedings of the 9th International Workshop on Dynamic Analysis*, WODA '11, pages 35–40, 2011.

[GVNH11] Benoit Gaudin, Emil Iordanov Vassev, Patrick Nixon, and Michael Hinchey. A control theory based approach for self-healing of un-handled runtime exceptions. In *Proceedings of the 8th ACM International Conference on Autonomic Computing*, ICAC '11, pages 217–220, 2011.

[GZ05] Sherif A. Gurguis and Amir Zeid. Towards autonomic web services: Achieving self-healing using web services. *ACM SIGSOFT Software Engineering Notes*, 30(4):1–5, 2005.

[HBS+12] Abram Hindle, Earl T. Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. On the naturalness of software. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 837–847, 2012.

[HC13] Petr Hosek and Cristian Cadar. Safe software updates via multi-version execution. In *Proceedings of the 35th International Conference on Software Engineering*, ICSE '13, pages 612–621, 2013.

[HCU+07] T. Harris, A. Cristal, O.S. Unsal, E. Ayguade, F. Gagliardi, B. Smith, and M. Valero. Transactional memory: An overview. *Micro, IEEE*, 27(3):8–29, 2007.

[HDO10] Shunsuke Hiroyama, Tadashi Dohi, and Hiroyuki Okamura. Comparison of aperiodic checkpoint placement algorithms. In G.S. Tomar, Ruay-Shiung Chang, Osvaldo Gervasi, Tai-hoon Kim, and SamirKumar Bandyopadhyay, editors, *Advanced Computer Science and Information Technology*, volume 74 of *Communications in Computer and Information Science*, pages 145–156. Springer Berlin Heidelberg, 2010.

[Her91] Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, 1991.

[Her09] Maurice Herlihy. Transactional memory today: A status report. In *Proceedings of the 13th International Conference on Principles of Distributed Systems*, OPODIS '09, page 1, 2009.

[HG06] Benjamin Hindman and Dan Grossman. Atomicity via source-to-source translation. In *Proceedings of the 2006 Workshop on Memory System Performance and Correctness*, MSPC '06, pages 82–91, 2006.

[HKKF95] Yennun Huang, Chandra Kintala, Nick Kolettis, and N. Dudley Fulton. Software rejuvenation: Analysis, module and applications. In *Proceedings of the 25th International Symposium on Fault-Tolerant Computing*, FTCS '95, page 381, 1995.

[HLM03] Maurice Herlihy, Victor Luchangco, and Mark Moir. Obstruction-free synchronization: Double-ended queues as an example. In *Proceedings of the 23rd International Conference on Distributed Computing Systems*, ICDCS '03, page 522, 2003.

[HLM06] Maurice Herlihy, Victor Luchangco, and Mark Moir. A flexible framework for implementing software transactional memory. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, OOPSLA '06, pages 253–262, 2006.

[HLMS03] Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer, III. Software transactional memory for dynamic-sized data structures. In *Proceedings of the 22nd Symposium on Principles of Distributed Computing*, PODC '03, pages 92–101, 2003.

[HLMSR74] James J. Horning, Hugh C. Lauer, P. M. Melliar-Smith, and Brian Randell. A program structure for error detection and recovery. In *Proceedings of an International Symposium on Operating Systems*, pages 171–187, 1974.

[HM93] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th International Symposium on Computer Architecture*, ISCA '93, pages 289–300, 1993.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12:576–580, 1969.

[HOB05] Reza A. Haydarlou, Benno J. Overeinder, and Frances M.T. Brazier. A self-healing approach for object-oriented applications. In *Proceedings of 16th International Workshop on Database and Expert Systems Applications*, DEXA '05, pages 191– 195, 2005.

[Hor01] Paul Horn. Autonomic computing: IBM's perspective on the state of information technology. Technical report, IBM Research, 2001.

[How98] Jon Howell. Straightforward java persistence through checkpointing. In *Proceedings of the 8th International Workshop on Persistent Object Systems and of the 3rd International Workshop on Persistence and Java: Advances in Persistent Object Systems*, POS-PJW '98, pages 322–334, 1998.

[IBM06] IBM. An architectural blueprint for autonomic computing. Technical report, IBM Research, 2006.

[IPW01] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight java: a minimal core calculus for java and gj. *ACM Transactions on Programming Languages and Systems*, 23(3):396–450, 2001.

[JH05] James A. Jones and Mary Jean Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the 20th International Conference on Automated Software Engineering*, ASE '05, pages 273–282, 2005.

[JMSG07] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, pages 96–105, 2007.

[JS09] Lingxiao Jiang and Zhendong Su. Automatic mining of functionally equivalent code fragments via random testing. In *Proceedings of the 18th International Symposium on Software Testing and Analysis*, ISSTA '09, pages 81–92, 2009.

[JSK11] Rene Just, Franz Schweiggert, and Gregory M. Kapfhammer. MAJOR: An efficient and extensible tool for mutation analysis in a java compiler. In *Proceedings of the 26th International Conference on Automated Software Engineering*, ASE '11, pages 612– 615, 2011.

[JZ88] David B. Johnson and Willy Zwaenepoel. Recovery in distributed systems using asynchronous message logging and checkpointing. In *Proceedings of the 7th Symposium on Principles of Distributed Computing*, PODC '88, pages 171–181, 1988.

[KC03] Jeffrey O. Kephart and David M. Chess. The vision of autonomic computing. *IEEE Computer*, 36:41–50, 2003.

[KKI02] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. Ccfinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(7):654–670, 2002.

[KM07] Jeff Kramer and Jeff Magee. Self-managed systems: an architectural challenge. In *Proceedings of the Future of Software Engineering*, FOSE '07, pages 259–268, 2007.

[KR81] H. T. Kung and John T. Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems*, 6(2):213–226, 1981.

[KSNM05] Miryung Kim, Vibha Sazawal, David Notkin, and Gail Murphy. An empirical study of code clone genealogies. In *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE '05, pages 187–196, 2005.

[KT86] Richard Koo and Sam Toueg. Checkpointing and rollback-recovery for distributed systems. In *Proceedings of 1986 ACM Fall Joint Computer Conference*, ACM '86, pages 1150–1158, 1986.

[LBK90] Jean-Claude Laprie, Christian Béounes, and Karama Kanoun. Definition and analysis of hardware- and software-fault-tolerant architectures. *Computer*, 23(7):39–51, 1990.

[LF90] C.-C.J. Li and W.K. Fuchs. Catch-compiler-assisted techniques for checkpointing. In *Proceedings of the 20th International Symposium on Fault-Tolerant Computing*, FTCS '90, pages 74–81, 1990.

[LGDVFW12] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 3–13, 2012.

[LM00] Julia L. Lawall and Gilles Muller. Efficient incremental checkpointing of java programs. In *Proceedings of the 30th International Conference on Dependable Systems and Networks*, DSN '00, pages 61–70, 2000.

[LMP08] Davide Lorenzoli, Leonardo Mariani, and Mauro Pezzè. Automatic generation of software behavioral models. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 501–510, 2008.

[LMP09] David Lo, Leonardo Mariani, and Mauro Pezzè. Automatic steering of behavioral model inference. In *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, ESEC/FSE '09, pages 345–354, 2009.

[LMX05] Nik Looker, Malcolm Munro, and Jie Xu. Increasing web service dependability through consensus voting. In *Proceedings of the 29th Annual International Computer Software and Applications Conference*, COMPSAC '05, pages 66–69, 2005.

[LS79] B. H. Liskov and A. Snyder. Exception handling in clu. *IEEE Transactions on Software Engineering*, 5(6):546–558, 1979.

[LS98] Jun Lang and David B. Stewart. A study of the applicability of existing exception-handling techniques to component-based real-time software technology. *ACM Transactions on Programming Languages and Systems*, 20(2):274–301, 1998.

[LTBL97] Michael Litzkow, Todd Tannenbaum, Jim Basney, and Miron Livny. Checkpoint and Migration of UNIX Processes in the Condor Distributed Processing System. Technical report, University of Wisconsin, Madison, 1997.

[Lyu95] Michael R. Lyu. *Software Fault Tolerance*. John Wiley & Sons, Inc., 1995.

[MB08] Adina Mosincat and Walter Binder. Transparent runtime adaptability for bpel processes. In *Proceedings of the 6th International Conference on Service-Oriented Computing*, ICSOC '08, pages 241–255, 2008.

[Mey92] Bertrand Meyer. Applying "design by contract". *IEEE Computer*, 25:40–51, 1992.

[Mey01] Scott Meyers. *Effective STL: 50 Specific Ways to Improve Your Use of the Standard Template Library*. Addison-Wesley Longman Ltd., 2001.

[MMP06] Stefano Modafferi, Enrico Mussi, and Barbara Pernici. Sh-bpel: a self-healing plug-in for ws-bpel engines. In *Proceedings of the 1st Workshop on Middleware for Service Oriented Computing*, MW4SOC '06, pages 48–53, 2006.

[NKHL02] H. Nam, J. Kim, S. J. Hong, and S. Lee. Probabilistic checkpointing. *IEICE Transactions, Information and Systems*, E85-D:1093–1104, 2002.

[Noe10] Cyprien Noel. Extensible software transactional memory. In *Proceedings of the 3rd C* Conference on Computer Science and Software Engineering*, C3S2E '10, pages 23–34, 2010.

[NvS90] Victor F. Nicola and J.M. van Spanje. Comparative analysis of different models of checkpointing and recovery. *IEEE Transactions on Software Engineering*, 16(8):807–821, 1990.

[OKFN97] Shunji Osaki, Naoto Kaio, Satoshi Fukumoto, and Sayori Nakagawa. Optimal checkpoint policies attending with unsuccessful rollback recovery. *International Journal of Reliability, Quality and Safety Engineering*, 4(4):427–439, 1997.

[PBK95] James S. Plank, Micah Beck, and Gerry Kingsley. Compiler-assisted memory exclusion for fast checkpointing. *IEEE Technical Committee on Operating Systems and Application Environments*, 7(4):10–14, 1995.

[PBKL95] James S. Plank, Micah Beck, Gerry Kingsley, and Kai Li. Libckpt: Transparent checkpointing under unix. In *Proceedings of the USENIX Technical Conference*, TCON '95, page 18, 1995.

[PCL+99] James S. Plank, Yuqun Chen, Kai Li, Micah Beck, and Gerry Kingsley. Memory exclusion: Optimizing the performance of checkpointing systems. *Software: Practice and Experience*, 29(2):125–142, 1999.

[PKL+09] Jeff H. Perkins, Sunghun Kim, Sam Larsen, Saman Amarasinghe, Jonathan Bachrach, Michael Carbin, Carlos Pacheco, Frank Sherwood, Stelios Sidiroglou, Greg Sullivan, Weng-Fai Wong, Yoav Zibin, Michael D. Ernst, and Martin Rinard. Automatically patching errors in deployed software. In *Proceedings of the 22nd ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '09, pages 87–102, 2009.

[PRRS01] Peter Popov, Steve Riddle, Alexander Romanovsky, and Lorenzo Strigini. On systematic design of protectors for employing OTS items. In *Proceedings of the 27th Euromicro Conference on Software Engineering and Advanced Applications*, Euromicro '01, pages 22–29, 2001.

[PW76] David Lorge Parnas and Harald Würges. Response to undesired events in software systems. In *Proceedings of the 2nd International Conference on Software Engineering*, ICSE '76, pages 437–446, 1976.

[PW09] Mauro Pezzè and Jochen Wuttke. Automatic generation of runtime failure detectors from property templates. In Betty H. C. Cheng, Rogério de Lemos, Holger Giese, Paola Inverardi, and Jeff Magee, editors, *Software Engineering for Self-Adaptive Systems*, volume 5525 of *Lecture Notes in Computer Science*, pages 223–240. Springer-Verlag, 2009.

[QTSZ05] Feng Qin, Joseph Tucek, Jagadeesan Sundaresan, and Yuanyuan Zhou. Rx: Treating bugs as allergies—a safe method to survive software failures. In *Proceedings of the 20th ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '05, pages 235–248, 2005.

[Ran75] Brian Randell. System structure for software fault tolerance. In *Proceedings of the International Conference on Reliable Software*, pages 437–449, 1975.

[RCK09] Chanchal K. Roy, James R. Cordy, and Rainer Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of Computer Programming*, 74(7):470–495, 2009.

[RN07] Lukas Renggli and Oscar Nierstrasz. Transactional memory for smalltalk. In *Proceedings of the 15th International Conference on Dynamic Languages*, ICDL '07, pages 207–221, 2007.

[RN09] Lukas Renggli and Oscar Nierstrasz. Transactional memory in a dynamic language. *Computer Languages, Systems and Structures*, 35(1):21–30, 2009.

[Rom02] Eric Roman. A survey of checkpoint/restart implementations. Technical report, Lawrence Berkeley National Laboratory, Tech, 2002.

[Ros95] David S. Rosenblum. A practical approach to programming with assertions. *IEEE Transactions on Software Engineering*, 21:19–31, 1995.

[RS95] Mark Russinovich and Zary Segall. Fault-tolerance for off-the-shelf applications and hardware. In *Proceedings of the 25th International Symposium on Fault-Tolerant Computing*, FTCS '95, pages 67–71, 1995.

[RW02] Algis Rudys and Dan S. Wallach. Transactional rollback for language-based systems. In *Proceedings of the 32th International Conference on Dependable Systems and Networks*, DSN '02, pages 439–448, 2002.

[Sah06] Goutam Kumar Saha. Software based fault tolerance: A survey. *Ubiquity*, 2006, 2006.

[SAM10] Hesam Samimi, Ei Darli Aung, and Todd Millstein. Falling back on executable specifications. In *Proceedings of the 24th European Conference on Object-Oriented Programming*, ECOOP '10, pages 552–576, 2010.

[Sco83] Roderick Keith Scott. *Data Domain Modeling of Fault-Tolerant Software Reliability*. PhD thesis, 1983.

[SG95] Ushio Sumita and Paulo B. Goes. Stochastic models for performance analysis of database recovery control. *IEEE Transactions on Computers*, 44(4):561–576, 1995.

[She08] O. Shehory. Shadows: Self-healing complex software systems. In *Proceedings of the 23th International Conference on Automated Software Engineering*, ASE '08, pages 71–76, 2008.

[SKAZ04] Sudarshan M. Srinivasan, Srikanth Kandula, Christopher R. Andrews, and Yuanyuan Zhou. Flashback: A lightweight extension for rollback and deterministic replay for software debugging. In *Proceedings of the Conference on USENIX Annual Technical Conference*, ATEC '04, page 3, 2004.

[SKG89] Ushio Sumita, Naoto Kaio, and PauloB. Goes. Analysis of effective service time with age dependent interruptions and its application to optimal rollback policy for database management. *Queueing Systems*, 4(3):193–212, 1989.

[SL86] Kang G. Shin and Yann-Hang Lee. Measurement and application of fault latency. *IEEE Transactions on Computers*, 35(4):370–375, 1986.

[Smi88] Jonathan M Smith. A survey of software fault tolerance techniques. Technical report, Department of Computer Science, Columbia University, 1988.

[SSG+09] Saurabh Sinha, Hina Shah, Carsten Görg, Shujuan Jiang, Mijung Kim, and Mary Jean Harrold. Fault localization and repair for java runtime exceptions. In *Proceedings of the 18th international Symposium on Software Testing and Analysis*, ISSTA '09, pages 153–164, 2009.

[ST95] Nir Shavit and Dan Touitou. Software transactional memory. In *Proceedings of the 14th Symposium on Principles of Distributed Computing*, PODC '95, pages 204–213, 1995.

[STN+08] Sattanathan Subramanian, Philippe Thiran, Nanjangud C. Narendra, Ghita Kouadri Mostefaoui, and Zakaria Maamar. On the enhancement of bpel engines for self-healing composite web services. In *Proceedings of the 8th International Symposium on Applications and the Internet*, SAINT '08, pages 33–39, 2008.

[Sue00] Takashi Suezawa. Persistent execution state of a java virtual machine. In *Proceedings of the Conference on Java Grande*, JAVA '00, pages 160–167, 2000.

[TBFM06] Yehia Taher, Djamal Benslimane, Marie-Christine Fauvet, and Zakaria Maamar. Towards an approach for web services substitution. In *Proceedings of the 10th International Database Engineering and Applications Symposium*, IDEAS '06, pages 166–173, 2006.

[TMB80] David J. Taylor, David E. Morgan, and James P. Black. Redundancy in data structures: Improving software fault tolerance. *IEEE Transactions on Software Engineering*, 6(6):585–594, 1980.

[Wey82] Elaine J. Weyuker. On testing non-testable programs. *The Computer Journal*, 25(4):465–470, 1982.

[WHV+95] Yi-Min Wang, Yennun Huang, Kiem-Phong Vo, Pe-Yu Chung, and C. Kintala. Checkpointing and its applications. In *Proceedings of the 25th International Symposium on Fault-Tolerant Computing*, FTCS '95, pages 22–31, 1995.

[WKII10] Long Wang, Zbigniew Kalbarczyk, Ravishankar K. Iyer, and Arun Iyengar. Checkpointing virtual machines against transient errors. In *Pro-*

ceedings of the 16th International On-Line Testing Symposium, IOLTS '10, pages 97–102, 2010.

[WM89] P. R. Wilson and T. G. Moher. Demonic memory for process histories. In *Proceedings of the 10th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '89, pages 330–343, 1989.

[WNLGF09] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. Automatically finding patches using genetic programming. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 364–374, 2009.

[WP96] Darrin West and Kiran Panesar. Automatic incremental state saving. In *Proceedings of the 10th Workshop on Parallel and Distributed Simulation*, PADS '96, pages 78–85, 1996.

[WPF+10] Yi Wei, Yu Pei, Carlo A. Furia, Lucas S. Silva, Stefan Buchholz, Bertrand Meyer, and Andreas Zeller. Automated fixing of programs with contracts. In *Proceedings of the 19th International Symposium on Software Testing and Analysis*, ISSTA '10, pages 61–72, 2010.

[WYCL11] Lu Wei, Zhu Yian, Ma Chunyan, and Zhang Longmei. A model driven approach for self-healing computing system. In *Proceedings of the 7th International Conference on Computational Intelligence and Security*, CIS '11, pages 185–189, 2011.

[XRTQ07] Guoqing Xu, Atanas Rountev, Yan Tang, and Feng Qin. Efficient checkpointing of java software using context-sensitive capture and replay. In *Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, ESEC/FSE '07, pages 85–94, 2007.

[YC75] S. S. Yau and R. C. Cheung. Design of self-checking software. In *Proceedings of the International Conference on Reliable Software*, pages 450–455, 1975.

[YNW+08] Richard M. Yoo, Yang Ni, Adam Welc, Bratin Saha, Ali-Reza Adl-Tabatabai, and S. Hsien-Hsin Lee. Kicking the tires of software transactional memory: Why the going gets tough. In *Proceedings of the 20th annual Symposium on Parallelism in Algorithms and Architectures*, SPAA '08, pages 265–274, 2008.

[You74] John W. Young. A first order approximation to the optimum checkpoint interval. *Communications of the ACM*, 17(9):530–531, 1974.

[YP08] Michal Young and Mauro Pezzè. *Software Testing and Analysis: Process, Principles and Techniques*. John Wiley & Sons, Inc., 2008.

[Yu98] Weider D. Yu. A software fault prevention approach in coding and root cause analysis. *Bell Labs Technical Journal*, 3(2):3–21, 1998.